Introducción a la programación con Python

Andrés Becerra Sandoval

Traducción y Adaptación del libro
''How to think like a computer scientist, learning with Python'',
escrito por:
Allen Downey
Jeffrey Elkner
Chris Meyers



Facultad de Ingeniería



Rector: Jorge Humberto Peláez, S.J.
Vicerrector Académico: Antonio de Roux
Vicerrector del Medio Universitario: Gabriel Jaime Pérez, S.J.

Facultad de Ingeniería
Decano Académico: Jorge Francisco Estela
Decana del Medio Universitario: Claudia Lucía Mora

Titulo: Introducción a la programación con Python
Titulo original: How to think like a computer scientist, learning with Python Autores: Allen Downey, Jeffrey Elkner, Chris Meyers
Traducción y adaptación: Andrés Becerra Sandoval
Colección: Libro

ISBN: 978-958-8347-22-6

Coordinador Editorial: Ignacio Murgueitio
Email: mignacio@javerianacali.edu.co

Correspondencia, suscripciones y solicitudes de canje:
Calle 18 # 118-250
Santiago de Cali, Valle del Cauca
Pontificia Universidad Javeriana
Facultad de Ingeniería
Teléfonos: (57-2) 3218200 Exts. 233 - 518 Fax 555 2823
Email: abecerra@javerianacali.edu.co

Formato 17 x 25 cms
Diseño e Impresión: Multimedios PUJ Cali

Diseño de Carátula: Patricia Mejía, basada en una imagen de Ken Manheimer
http://myriadicity.net

Impresión: 2009

La forma original de este libro es código fuente LaTeX y compilarlo tiene el efecto de generar un libro de texto en una repesentacion independiente del dispositivo que puede ser convertida a otros formatos e imprimirse.

El código fuente LaTeX para este libro y mas información sobre este proyecto se encuentra en los sitios web:

```
http://cic.puj.edu.co/~abecerra
http://www.thinkpython.com
```

Este libro ha sido preparado utilizando LaTeX y las figuras se han realizado con xfig. Todos estos son programas de código abierto, gratuito.

Downey, Allen
Introducción a la programación con Python / Allen Downey, Jeffrey Elkner, Chris Meyers; traducido y adaptado por Andrés Becerra Sandoval. – Santiago de Cali: Pontificia Universidad Javeriana, Sello Editorial Javeriano, 2009.
305 p. ; 26 cm.

Incluye referencias bibliográficas e índice.

ISBN 978-958-8347-22-6

1. Programación (computadores electrónicos) – Metodología 2. Python (lenguaje de programación para computadores) I. Meyer, Chris II. Pontificia Universidad Javeriana (Cali) III. How to think like a computer scientist: learning with python IV. Tít.

SCDD 005.1 BPUJC

Prólogo

Por David Beazley

Como un educador, investigador y autor de libro, estoy encantado de ver la terminación de este texto. Python es un lenguaje de programación divertido y extremadamente fácil de usar, que ha ganado renombre constantemente en los años recientes. Desarrollado hace diez años por Guido van Rossum, la sintaxis simple de Python y su "sabor" se derivan, en gran parte del ABC, un lenguaje de programación para enseñanza que se desarrolló en los 1980s. Sin embargo, Python también fue creado para resolver problemas reales y tiene una amplia gama de características que se encuentran en lenguajes de programación como C++, Java, Modula-3, y Scheme. Debido a esto, uno de las características notables de Python es la atracción que ejerce sobre programadores profesionales, científicos, investigadores, artistas y educadores.

A pesar de ésta atracción en muchas comunidades diversas, usted puede todavía preguntarse "¿porque Python?" o "¿porque enseñar programación con Python?" Responder éstas preguntas no es una tarea fácil— especialmente cuando la opinión popular está del lado masoquista de usar alternativas como C++ y Java. Sin embargo, pienso que la respuesta mas directa es que la programación en Python es simplemente mas divertida y mas productiva.

Cuando enseño cursos de informática, yo quiero cubrir conceptos importantes, hacer el material interesante y enganchar a los estudiantes. Desafortunadamente, hay una tendencia en la que los cursos de programación introductorios dedican demasiada atención en la abstracción matemática y a hacer que los estudiantes se frustren con problemas molestos relacionados con la sintaxis, la compilación y la presencia de reglas arcanas en los lenguajes. Aunque la abstracción y el formalismo son importantes para los ingenieros de software y para los estudiantes de ciencias de la computación, usar este enfoque hace a la informática muy aburrida. Cuando enseño un curso no quiero tener un grupo de estudiantes sin inspiración. Quisiera verlos intentando resolver problemas interesantes, explorando ideas diferentes, intentando enfoques no convencionales, rompiendo reglas y aprendiendo de sus errores. En el proceso no quiero perder la mitad del semestre tratando de resolver problemas sintácticos oscuros, interpretando mensajes de error del compilador in-

comprensibles, o descifrando cuál de las muchas maneras en que un programa puede generar un error grave de memoria se está presentando.

Una de las razones del por qué me gusta Python es que proporciona un equilibrio muy bueno entre lo práctico y lo conceptual. Puesto que se interpreta Python, los principiantes pueden empezar a hacer cosas interesantes casi de inmediato sin perderse en problemas de compilación y enlace. Además, Python viene con una biblioteca grande de módulos, que pueden ser usados en dominios que van desde programación en la web hasta gráficos. Tener un foco práctico es una gran manera de enganchar a los estudiantes y permite que emprendan proyectos significativos. Sin embargo, Python también puede servir como una excelente base para introducir conceptos importantes de la informática. Puesto que Python soporta completamente procedimientos y clases, los estudiantes pueden ser introducidos gradualmente a temas como la abstracción procedimental, las estructuras de datos y la programación orientada a objetos—lo que se puede aplicar después a cursos posteriores en Java o C++. Python proporciona, incluso, varias características de los lenguajes de programación funcionales y puede usarse para introducir conceptos que se pueden explorar con más detalle en cursos con Scheme y Lisp.

Leyendo, el prefacio de Jeffrey, estoy sorprendido por sus comentarios de que Python le permita ver un "más alto nivel de éxito y un nivel bajo de frustración" y que puede "avanzar mas rápido con mejores resultados." Aunque estos comentarios se refieren a sus cursos introductorios, a veces uso Python por estas mismas razones en los cursos de informática avanzada en la Universidad de Chicago. En estos cursos enfrento constantemente la tarea desalentadora de cubrir un montón de material difícil durante nueve semanas. Aunque es totalmente posible para mi infligir mucho dolor y sufrimiento usando un lenguaje como C++, he encontrado a menudo que este enfoque es improductivo—especialmente cuando el curso se trata de un asunto sin relación directa con la "programación." He encontrado que usar Python me permite enfocar el tema del curso y dejar a los estudiantes desarrollar proyectos substanciales.

Aunque Python siga siendo un lenguaje joven y en desarrollo, creo que tiene un futuro brillante en la educación. Este libro es un paso importante en esa dirección.

David Beazley
Universidad de Chicago, Autor de *Python Essential Reference*

Prefacio

Por Jeff Elkner
Este libro debe su existencia a la colaboración hecha posible por Internet y el movimiento de software libre. Sus tres autores—un profesor de colegio, un profesor de secundaria y un programador profesional—tienen todavía que verse cara a cara, pero han podido trabajar juntos y han sido ayudados por maravillosas personas, quienes han donado su tiempo y energía para ayudar a hacer ver mejor este libro.
Nosotros pensamos que este libro es un testamento a los beneficios y futuras posibilidades de esta clase de colaboración, el marco que se ha puesto en marcha por Richard Stallman y el movimiento de software libre.

Cómo y porqué vine a utilizar Python

En 1999, el examen del College Board's Advanced Placement (AP) de Informática se hizo en C++ por primera vez. Como en muchas escuelas de Estados Unidos, la decisión para cambiar el lenguaje tenía un impacto directo en el plan de estudios de informática en la escuela secundaria de Yorktown en Arlington, Virginia, donde yo enseño. Hasta este punto, Pascal era el lenguaje de instrucción en nuestros cursos del primer año y del AP. Conservando la práctica usual de dar a los estudiantes dos años de exposición al mismo lenguaje, tomamos la decisión de cambiar a C++ en el curso del primer año durante el periodo escolar 1997-98 de modo que siguiéramos el cambio del College Board's para el curso del AP el año siguiente.
Dos años después, estoy convencido de que C++ no era una buena opción para introducir la informática a los estudiantes. Aunque es un lenguaje de programación de gran alcance, también es extremadamente difícil de aprender y de enseñar. Me encontré constantemente peleando con la sintaxis difícil de C++ y sus múltiples maneras de hacer las cosas, y estaba perdiendo muchos estudiantes, innecesariamente, como resultado. Convencido de que tenía que haber una mejor opción para nuestras clases de primer año, fui en busca de un alternativa a C++.
Necesitaba un lenguaje que pudiera correr en las máquinas en nuestro laboratorio Linux, también en las plataformas de Windows y Macintosh, que muchos de los

estudiantes tienen en casa. Quería que fuese un lenguaje de código abierto, para que los estudiantes lo pudieran usar en casa sin pagar por una licencia. Quería un lenguaje usado por programadores profesionales, y que tuviera una comunidad activa alrededor de él. Tenía que soportar la programación procedural y orientada a objetos. Y más importante, tenía que ser fácil de aprender y de enseñar. Cuando investigué las opciones con estas metas en mente, Python saltó como el mejor candidato para la tarea.

Pedí a uno de los estudiantes más talentosos de Yorktown, Matt Ahrens, que le diera a Python una oportunidad. En dos meses él no sólo aprendió el lenguaje, sino que escribió una aplicación llamada pyTicket que permitió a nuestro personal atender peticiones de soporte tecnológico vía web. Sabia que Matt no podría terminar una aplicación de esa escala en tan poco tiempo con C++, y esta observación, combinada con el gravamen positivo de Matt sobre Python, sugirió que este lenguaje era la solución que buscaba.

Encontrando un libro de texto

Al decidir utilizar Python en mis dos clases de informática introductoria para el año siguiente, el problema más acuciante era la carencia de un libro.

El contenido libre vino al rescate. A principios del año, Richard Stallman me presentó a Allen Downey. Los dos habíamos escrito a Richard expresando interés en desarrollar un contenido gratis y educativo. Allen ya había escrito un libro de texto para el primer año de informática, *Como pensar como un científico de la computación*. Cuando leí este libro, inmediatamente quise usarlo en mi clase. Era el texto más claro y mas provechoso de introducción a la informática que había visto. Acentúa los procesos del pensamiento implicados en la programación más bien que las características de un lenguaje particular. Leerlo me hizo sentir un mejor profesor inmediatamente. *Como pensar como un científico de la computación con Java* no solo es un libro excelente, sino que también había sido publicado bajo la licencia publica GNU, lo que significa que podría ser utilizado libremente y ser modificado para resolver otras necesidades. Una vez que decidí utilizar Python, se me ocurrió que podía traducir la versión original del libro de Allen (en Java) al nuevo lenguaje (Python). Aunque no podía escribir un libro de texto solo, tener el libro de Allen me facilitó la tarea, y al mismo tiempo demostró que el modelo cooperativo usado en el desarrollo de software también podía funcionar para el contenido educativo.

Trabajar en este libro, por los dos últimos años, ha sido una recompensa para mis estudiantes y para mí; y mis estudiantes tuvieron una gran participación en el proceso. Puesto que podía realizar cambios inmediatos, siempre que alguien encontrara un error de deletreo o un paso difícil, yo les animaba a que buscaran errores en el libro, dándoles un punto cada vez que hicieran una sugerencia que resultara en un cambio en el texto. Eso tenía la ventaja doble de animarles a que leyeran el

texto más cuidadosamente y de conseguir la corrección del texto por sus lectores críticos más importantes, los estudiantes usándolo para aprender informática.
Para la segunda parte del libro, enfocada en la programación orientada a objetos, sabía que alguien con más experiencia en programación que yo era necesario para hacer el trabajo correctamente. El libro estuvo incompleto la mayoría del año hasta que la comunidad de software abierto me proporcionó de nuevo los medios necesarios para su terminación.
Recibí un correo electrónico de Chris Meyers, expresando su interés en el libro. Chris es un programador profesional que empezó enseñando un curso de programación el año anterior, usando Python en el Lane Community College en Eugene, Oregon. La perspectiva de enseñar el curso llevó a Chris al libro, y él comenzó a ayudarme inmediatamente. Antes del fin de año escolar él había creado un proyecto complementario en nuestro Sitio Web `http://www.ibiblio.org/obp`, titulado *Python for Fun* y estaba trabajando con algunos de mis estudiantes más avanzados como profesor principal, guiándolos mas allá de donde yo podía llevarlos.

Introduciendo la programación con Python

El proceso de uso y traducción de *Como pensar como un científico de la computación*, por los últimos dos años, ha confirmado la conveniencia de Python para enseñar a estudiantes principiantes. Python simplifica bastante los ejemplos de programación y hace que las ideas importantes sean más fáciles de enseñar.
El primer ejemplo del texto ilustra este punto. Es el tradicional "hola, mundo", programa que en la versión C++ del libro se ve así:

```
#include <iostream.h>

void main()
{
  cout << "Hola, mundo." << endl;
}
```

en la versión Python es:

```
print "Hola, Mundo!"
```

Aunque este es un ejemplo trivial, las ventajas de Python salen a la luz. El curso de Informática I, en Yorktown, no tiene prerrequisitos, es por eso que muchos de los estudiantes, que ven este ejemplo, están mirando a su primer programa. Algunos de ellos están un poco nerviosos, porque han oído que la programación de computadores es difícil de aprender. La versión C++ siempre me ha forzado a escoger

entre dos opciones que no me satisfacen: explicar el `#include`, `void main()`, y las sentencias {, y } y arriesgar a confundir o intimidar a algunos de los estudiantes al principio, o decirles, "No te preocupes por todo eso ahora; lo retomaré más tarde," y tomar el mismo riesgo. Los objetivos educativos en este momento del curso son introducir a los estudiantes la idea de sentencia y permitirles escribir su primer programa. Python tiene exactamente lo que necesito para lograr esto, y nada más.

Comparar el texto explicativo de este programa en cada versión del libro ilustra más de lo que esto significa para los estudiantes principiantes. Hay trece párrafos de explicación de "Hola, mundo!" en la versión C++; en la versión Python, solo hay dos. Aún mas importante, los 11 párrafos que faltan no hablan de "grandes ideas" en la programación de computadores, sino de minucias de la sintaxis de C++. Encontré la misma situación al repasar todo el libro. Párrafos enteros desaparecían en la versión Python del texto, porque su sencilla sintaxis los hacía innecesarios.

Usar un lenguaje de muy alto nivel, como Python, le permite a un profesor posponer los detalles de bajo nivel de la máquina hasta que los estudiantes tengan el bagaje que necesitan para entenderlos. Permite "poner cosas primero" pedagógicamente. Unos de los mejores ejemplos de esto es la manera en la que Python maneja las variables. En C++ una variable es un nombre para un lugar que almacena una cosa. Las variables tienen que ser declaradas con tipos, al menos parcialmente, porque el tamaño del lugar al cual se refieren tiene que ser predeterminado. Así, la idea de una variable se liga con el hardware de la máquina. El concepto poderoso y fundamental de variable ya es difícil para los estudiantes principiantes (de informática y álgebra). Bytes y direcciones de memoria no ayudan para nada. En Python una variable es un nombre que se refiere a una cosa. Este es un concepto más intuitivo para los estudiantes principiantes y está más cerca del significado de "variable" que aprendieron en los cursos de matemática del colegio. Yo me demoré menos tiempo ayudándolos con el concepto de variable y en su uso este año, que en el pasado.

Otro ejemplo de cómo Python ayuda en la enseñanza y aprendizaje de la programación es su sintaxis para las funciones. Mis estudiantes siempre han tenido una gran dificultad comprendiendo las funciones. El problema principal se centra alrededor de la diferencia entre una definición de función y un llamado de función, y la distinción relacionada entre un parámetro y un argumento. Python viene al rescate con una bella sintaxis. Una definición de función empieza con la palabra clave `def`, y simplemente digo a mis estudiantes: "cuando definas una función, empieza con `def`, seguido del nombre de la función que estás definiendo, cuando llames una función, simplemente llama (digita) su nombre." Los parámetros van con las definiciones y los argumentos van con los llamados. No hay tipos de retorno, tipos para los parámetros, o pasos de parámetro por referencia y valor, y ahora yo puedo enseñar funciones en la mitad de tiempo que antes, con una mejor comprensión.

Usar Python ha mejorado la eficacia de nuestro programa de informática para todos los estudiantes. Veo un nivel general de éxito alto y un nivel bajo de la frustra-

ción, que ya había experimentado durante los dos años que enseñé C++. Avanzo más rápido y con mejores resultados. Más estudiantes terminan el curso con la habilidad de crear programas significativos; esto genera una actitud positiva hacia la experiencia de la programación.

Construyendo una comunidad

He recibido correos electrónicos de todas partes del mundo, de personas que están usando este libro para aprender o enseñar programación. Una comunidad de usuarios ha comenzado a emerger, y muchas personas han contribuido al proyecto mandando materiales a través del sitio Web complementario:

`http://www.thinkpython.com`.

Con la publicación del libro, en forma impresa, espero que continúe y se acelere el crecimiento de esta comunidad de usuarios.
La emergencia de esta comunidad y la posibilidad que sugiere para otras experiencias de colaboración similar entre educadores han sido las partes más excitantes de trabajar en este proyecto, para mí. Trabajando juntos, nosotros podemos aumentar la calidad del material disponible para nuestro uso y ahorrar tiempo valioso.
Yo les invito a formar parte de nuestra comunidad y espero escuchar de ustedes. Por favor escriba a los autores a `feedback@thinkpython.com`.

Jeffrey Elkner
Escuela Secundaria Yortown
Arlington, Virginia.

Lista de los colaboradores

Este libro vino a la luz debido a una colaboración que no sería posible sin la licencia de documentación libre de la GNU (Free Documentation License). Quisiéramos agradecer a la Free Software Foundation por desarrollar esta licencia y, por supuesto, por ponerla a nuestra disposición.
Nosotros queremos agradecer a los mas de 100 juiciosos y reflexivos lectores que nos han enviado sugerencias y correcciones durante los años pasados. En el espíritu del software libre, decidimos expresar nuestro agradecimiento en la forma de una lista de colaboradores. Desafortunadamente, esta lista no está completa, pero estamos haciendo nuestro mejor esfuerzo para mantenerla actualizada.
Si tiene la oportunidad de leer la lista, tenga en cuenta que cada persona mencionada aquí le ha ahorrado a usted y a todos los lectores subsecuentes la confusión debida a un error técnico o debido a una explicación confusa, solo por enviarnos una nota.
Después de tantas correcciones, todavía pueden haber errores en este libro. Si ve uno, esperamos que tome un minuto para contactarnos. El correo electrónico es `feedback@thinkpython.com`. Si hacemos un cambio debido a su sugerencias, usted aparecerá en la siguiente versión de la lista de colaboradores (a menos que usted pida ser omitido). Gracias!

- Lloyd Hugh Allen remitió una corrección a la Sección 8.4.
- Yvon Boulianne corrigió un error semántico en el Capítulo 5.
- Fred Bremmer hizo una corrección en la Sección 2.1.
- Jonah Cohen escribió los guiones en Perl para convertir la fuente LaTeX, de este libro, a un maravilloso HTML.
- Michael Conlon remitió una corrección de gramática en el Capítulo 3 una mejora de estilo en el Capítulo 2, e inició la discusión de los aspectos técnicos de los intérpretes.
- Benoit Girard envió una corrección a un extraño error en la Sección 5.6.

- Courtney Gleason y Katherine Smith escribieron `horsebet.py`, que se usaba como un caso de estudio en una versión anterior de este libro. Su programa se puede encontrar en su website.
- Lee Harr sometió más correcciones de las que tenemos espacio para enumerar aquí, y, por supuesto, debería ser listado como uno de los editores principales del texto.
- James Kaylin es un estudiante usando el texto. Él ha enviado numerosas correcciones.
- David Kershaw arregló la función errónea `imprimaDoble` en la Sección 3.10.
- Eddie Lam ha enviado numerosas correcciones a los Capítulos 1, 2, y 3. Él corrigió el Makefile para que creara un índice, la primera vez que se compilaba el documento, y nos ayudó a instalar un sistema de control de versiones.
- Man-Yong Lee envió una corrección al código de ejemplo en la Sección 2.4.
- David Mayo notó que la palabra “inconscientemente”debe cambiarse por “subconscientemente”.
- Chris McAloon envió varias correcciones a las Secciones 3.9 y 3.10.
- Matthew J. Moelter ha sido un contribuidor de mucho tiempo quien remitió numerosas correcciones y sugerencias al libro.
- Simon Dicon Montford reportó una definición de función que faltaba y varios errores en el Capítulo 3. Él también encontró errores en la función `incrementar` del Capítulo 13.
- John Ouzts corrigió la definición de “valor de retorno” en el Capítulo 3.
- Kevin Parks envió sugerencias valiosas para mejorar la distribución del libro.
- David Pool envió la corrección de un error en el glosario del Capítulo 1 y palabras de estímulo.
- Michael Schmitt envió una corrección al capítulo de archivos y excepciones.
- Robin Shaw notó un error en la Sección 13.1, donde la función imprimirHora se usaba en un ejemplo sin estar definida.
- Paul Sleigh encontró un error en el Capítulo 7 y otro en los guiones de Perl de Jonah Cohen que generan HTML a partir de LaTeX.

- Craig T. Snydal está probando el texto en un curso en Drew University. El ha aportado varias sugerencias valiosas y correcciones.
- Ian Thomas y sus estudiantes están usando el texto en un curso de programación. Ellos son los primeros en probar los capítulos de la segunda mitad del libro y han enviado numerosas correcciones y sugerencias.
- Keith Verheyden envió una corrección al Capítulo 3.
- Peter Winstanley descubrió un viejo error en nuestro Latín, en el capítulo 3.
- Chris Wrobel hizo correcciones al código en el capítulo sobre archivos, E/S y excepciones.
- Moshe Zadka hizo contribuciones inestimables a este proyecto. Además de escribir el primer bosquejo del capítulo sobre Diccionarios, también proporcionó una dirección continua en los primeros años del libro.
- Christoph Zwerschke envió varias correcciones y sugerencias pedagógicas, y explicó la diferencia entre *gleich* y *selbe*.
- James Mayer nos envió una ciénaga entera de errores tipográficos y de deletreo, incluyendo dos en la lista de colaboradores
- Hayden McAfee descubrió una inconsistencia potencialmente confusa entre dos ejemplos.
- Angel Arnal hace parte de un equipo internacional de traductores que trabajan en la versión española del texto. Él también ha encontrado varios errores en la versión inglesa.

Traducción al español

Al comienzo de junio de 2007 tomé la iniciativa de traducir el texto "How to think like a Computer Scientist, with Python" al español. Rápidamente me dí cuenta de que ya había un trabajo inicial de traducción empezado por:

- Angel Arnal
- I Juanes
- Litza Amurrio
- Efrain Andia

Ellos habían traducido los capítulos 1,2,10,11, y 12, así como el prefacio, la introducción y la lista de colaboradores. Tomé su valioso trabajo como punto de partida, adapté los capítulos, traduje las secciones faltantes del libro y añadí un primer capítulo adicional sobre solución de problemas.

Para realizar este trabajo ha sido invaluable la colaboración de familiares, colegas, amigos y estudiantes que han señalado errores, expresiones confusas y han aportado toda clase de sugerencias constructivas. Mi agradecimiento va para los traductores antes mencionados y para:

- Beatriz Eugenia Marín, mi esposa, quien encontró varios errores en el texto.
- Los profesores que dictan el curso "Introducción a la Programación", en la Pontificia Universidad Javeriana (Cali-Colombia):
    - Mario Julián Mora

Andrés Becerra Sandoval
Pontificia Universidad Javeriana - Seccional Cali
abecerra@cic.puj.edu.co

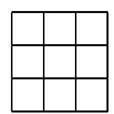
Índice general

Capítulo 1

Solución de problemas

1.1. Solución de acertijos

Todos nos hemos topado con acertijos como el siguiente. Disponga los dígitos del 1 al 9 en el recuadro siguiente, de manera que la suma de cada fila, cada columna y las dos diagonales dé el mismo resultado:



Este acertijo se denomina construcción de un **cuadrado mágico**. Un acertijo, normalmente, es un enigma o adivinanza que se propone como pasatiempo. Otros ejemplos de acertijo son un crucigrama, una sopa de letras y un sudoku.
Los acertijos pueden tener varias soluciones, por ejemplo, la siguiente es una solución propuesta al acertijo anterior:

1	2	3
4	5	6
7	8	9

Usted puede notar que esta solución candidata no es correcta. Si tomamos la suma por filas, obtenemos valores distintos:

- En la fila 1: 1+2+3=6
- En la fila 2: 4+5+6=15
- En la fila 3: 7+8+9=24

Si tomamos las columnas, tampoco obtenemos el mismo resultado en cada suma:

- En la columna 1: 1+4+7=12
- En la columna 2: 2+5+8=15
- En la columna 3: 3+6+9=18

A pesar de que las diagonales sí suman lo mismo:

- En la diagonal 1: 1+5+9=15
- En la diagonal 2: 7+5+3=15

Tómese un par de minutos para resolver este acertijo, es decir, para construir un cuadrado mágico y regrese a la lectura cuando obtenga la solución.
Ahora, responda para sí mismo las siguientes preguntas:

- ¿Cuál es la solución que encontró?
- ¿Es correcta su solución?
- ¿Cómo le demuestra a alguien que su solución es correcta?
- ¿Cuál fue el proceso de solución que llevo a cabo en su mente?
- ¿Cómo le explicaría a alguien el proceso de solución que llevó a cabo?
- ¿Puede poner por escrito el proceso de solución que llevó a cabo?

El reflexionar seriamente sobre estas preguntas es muy importante, tenga la seguridad de que esta actividad será muy importante para continuar con la lectura.
Vamos a ir contestando las preguntas desde una solución particular, y desde un proceso de solución particular, el de los autores. Su solución y su proceso de solución son igualmente valiosos, el nuestro solo es otra alternativa; es más, puede que hayamos descubierto la misma:

4	9	2
3	5	7
8	1	6

Esta solución es correcta, porque la suma por filas, columnas y de las dos diagonales da el mismo valor, 15. Ahora, para demostrarle a alguien este hecho podemos revisar las sumas por filas, columnas y diagonales detalladamente.
El proceso de solución que llevamos a cabo fue el siguiente:

- Sospechábamos que el 5 debía estar en la casilla central, ya que es el número medio de los 9: 1 2 3 4 **5** 6 7 8 9.

- Observamos un patrón interesante de la primera solución propuesta: las diagonales sumaban igual, 15:

1	2	**3**
4	**5**	6
7	8	**9**

- La observación anterior, 1+5+9=7+5+3, también permite deducir otro hecho interesante. Como el 5 está en las dos sumas, podemos deducir que 1+9=7+3, y esto es 10.

- Una posible estrategia consiste en colocar parejas de números que sumen 10, dejando al 5 "emparedado", por ejemplo, poner la pareja 6,4:

	6	
	5	
	4	

- Para agilizar ésta estrategia es conveniente enumerar todas las parejas de números entre 1 y 9 que suman 10, excluyendo al 5: (1,9),(2,8),(3,7),(4,6)

- Ahora, podemos probar colocando éstas parejas en filas, columnas y diagonales.

- Un primer ensayo:

7	**6**	2
	5	
8	**4**	3

  Aquí encontramos que es imposible armar el cuadrado, pues no hay como situar el 9 ni el 1. Esto sugiere que la pareja (6,4) no va en la columna central, debemos cambiarla.

- Después de varios ensayos, moviendo la pareja (6,4), y reacomodando los otros números logramos llegar a la solución correcta.

1.2. El método de solución

Mas allá de la construcción de cuadrados mágicos, lo que el ejemplo anterior pretende ilustrar es la importancia que tiene el usar un método ordenado de solución de problemas, acertijos en este caso.
Observe que la solución anterior fue conseguida a través de varios pasos sencillos, aunque el encadenamiento de todos éstos hasta producir un resultado correcto pueda parecer algo complejo.
Lea cualquier historia de Sherlock Holmes y obtendrá la misma impresión. Cada caso, tomado por el famoso detective, presenta un enigma difícil, que al final es resuelto por un encadenamiento de averiguaciones, deducciones, conjeturas y pruebas sencillas. Aunque la solución completa de cada caso demanda un proceso complejo de razonamiento, cada paso intermedio es sencillo y está guiado por preguntas sencillas y puntuales.
Las aventuras de Sherlock Holmes quizás constituyen una de las mejores referencias bibliográficas en el proceso de solución de problemas. Son como los capítulos de C.S.I[1], puestos por escrito. Para Holmes, y quizás para Grissom, existen varios principios que se deben seguir:

- Obtenerr las soluciones y quedarse satisfecho con ellas. El peor inconveniente para consolidar un método de solución de problemas poderoso consiste en que cuando nos topemos con una solución completa de un solo golpe nos quedemos sin reflexionar cómo llegamos a ella. Así, nunca aprenderemos estrategias valiosas para resolver el siguiente problema. Holmes decía que nunca adivinaba, primero recolectaba la mayor información posible sobre los problemas que tenía a mano, de forma que estos datos le "sugirieran" alguna solución.

- Todo empieza por la observación, un proceso minucioso de recolección de datos sobre el problema a mano. Ningún dato puede ignorarse de entrada, hasta que uno tenga una comprensión profunda sobre el problema.

- Hay que prepararse adecuadamente en una variedad de dominios. Para Holmes esto iba desde tiro con pistola, esgrima, boxeo, análisis de huellas, entre otros. Para cada problema que intentemos resolver habrá un dominio en el cual debemos aprender lo más que podamos, esto facilitará enormemente el proceso de solución.

- Prestar atención a los detalles "inusuales". Por ejemplo, en nuestro cuadrado mágico fue algo inusual que las diagonales de la primera alternativa de solución, la más ingenua posible, sumaran lo mismo.

[1] http://www.cbs.com/primetime/csi/

- Para deducir más hechos, a partir de los datos recolectados y los detalles inusuales, hay que usar todas las armas del razonamiento: el poder de la deducción (derivar nuevos hechos a partir de los datos), la inducción (generalizar a partir de casos), la refutación (el proceso de probar la falsedad de alguna conjetura), el pensamiento analógico (encontrando relaciones,metáforas, analogías) y, por último, pero no menos importante, el uso del sentido común.

- Después del análisis de datos, uno siempre debe proponer una alternativa de solución, así sea simple e ingenua, y proceder intentando probarla y **refutarla** al mismo tiempo. Vale la pena recalcar esto: no importa que tan ingenua, sencilla e incompleta es una alternativa de solución, con tal de que nos permita seguir indagando. Esto es como la primera frase que se le dice a una chica (o chico, dado el caso) que uno quiere conocer; no importa qué frase sea, no importa que tan trivial sea, con tal de que permita iniciar una conversación.

- El proceso de búsqueda de soluciones es como una conversación que se inicia, aunque en este caso el interlocutor no es una persona, sino el problema que tenemos a mano. Con una primera alternativa de solución—no importa lo sencilla e incompleta— podemos formularnos una pregunta interesante: ¿resuelve esta alternativa el problema?

- Lo importante de responder la pregunta anterior no es la obtención de un **No** como respuesta; pues esto es lo que sucede la mayoría de las veces. Lo importante viene cuando nos formulamos esta segunda pregunta ¿Con lo que sabemos del problema hasta ahora **por qué mi alternativa no es capaz de resolverlo?**

- La respuesta a la pregunta anterior puede ser: todavía no sé lo suficiente sobre el problema para entender por qué mi alternativa de solución no lo resuelve; esto es una señal de alerta para recolectar más datos y estudiar mas el dominio del problema.

- Una respuesta más constructiva a la pregunta anterior puede ser: mi alternativa de solución no resuelve el problema porque no considera algunos hechos importantes, y no considera algunas restricciones que debe cumplir una solución. Un ejemplo de este tipo de respuesta lo da nuestro ensayo de colocar la pareja (6,4) emparedando al 5 en el problema del cuadrado mágico:

7	**6**	2
	5	
8	**4**	3

Cuando notamos que la pareja (6,4) no puede colocarse en la columna central del cuadrado, intentamos otra alternativa de solución, colocando estos números en filas o en las esquinas. Lo importante de este tipo de respuesta es que nos va a permitir avanzar a *otra* alternativa de solución, casi siempre más compleja y más cercana a la solución.

- No poner obstáculos a la creatividad. Es muy difícil lograr esto porque la mente humana siempre busca límites para respetar; así, hay que realizar un esfuerzo consciente para eliminar todo límite o restricción que nuestra mente va creando. Una estrategia interesante es el uso y fortalecimiento del pensamiento lateral.

- Perseverar. El motivo más común de fracaso en la solución de acertijos y problemas es el abandono. No existe una receta mágica para resolver problemas, lo único que uno puede hacer es seguir un método y perseverar, perseverar sin importar cuantas alternativas de solución incorrectas se hayan generado. Esta es la clave para el éxito. Holmes decía que eran muchísimos más los casos que no había podido resolver, quizás Grissom reconocería lo mismo. Lo importante entonces es perseverar ante cada nuevo caso, esta es la única actitud razonable para enfrentar y resolver problemas.

1.3. Reflexión sobre este método de solución

El proceso de solución de problemas que hemos presentado es muy sencillo. Los grandes profesionales que tienen mucho éxito en su campo (científicos, humanistas, ingenieros, médicos, empresarios, etc.) tienen métodos de solución de problemas mucho más avanzados que el que hemos presentado aquí. Con esto pretendemos ilustrar un punto: lo importante no es el método propuesto, porque muy probablemente no le servirá para resolver todos los problemas o acertijos que enfrente; lo importante es:

- Contar con un método de solución de problemas. Si no hay método, podemos solucionar problemas, claro está, pero cada vez que lo hagamos será por golpes de suerte o inspiración –que no siempre nos acompaña, desafortunadamente.

- Desarrollar, a medida que transcurra el tiempo y adquiera más conocimientos sobre su profesión y la vida, un método propio de solución de problemas. Este desarrollo personal puede tomar como base el método expuesto aquí, o algún otro que encuentre en la literatura o a través de la interacción con otras personas.

1.4. Acertijos propuestos

Para que empiece inmediatamente a construir su método personal de solución de problemas, tome cada uno de los siguientes acertijos, siga el proceso de solución recomendado y documéntelo a manera de entrenamiento. Si usted descubre estrategias generales que no han sido consideradas aquí, compártalas con sus compañeros, profesores y, mejor aún, con los autores del libro.

1. Considere un tablero de ajedrez de 4x4 y 4 damas del mismo color. Su misión es colocar las 4 damas en el tablero sin que éstas se ataquen entre sí. Recuerde que una dama ataca a otra ficha si se encuentran en la misma fila, columna o diagonal.

2. Partiendo de la igualdad $a = b$, encuentre cuál es el problema en el siguiente razonamiento:

$$\begin{aligned} a &= b \\ a^2 &= ba \\ a^2 - b^2 &= ba - b^2 \\ (a-b)(a+b) &= b(a-b) \\ a+b &= b \\ a &= 2b \\ \frac{a}{b} &= 2 \\ 1 &= 2 \end{aligned}$$

   Tenga en cuenta que en el último paso se volvió a usar el hecho de que $a = b$, en la forma $\frac{a}{b} = 1$.

3. Encuentre el menor número entero positivo que pueda descomponerse como la suma de los cubos de dos números enteros positivos de dos maneras distintas. Esto es, encontrar el mínimo A tal que $A = b^3 + c^3$ y $A = d^3 + e^3$, con A, b, c, d y e números positivos, mayores a cero, y distintos entre si.

1.5. Mas allá de los acertijos: problemas computacionales

Un problema computacional es parecido a un acertijo; se presenta una situación problemática y uno debe diseñar alguna solución. En los problemas computacionales la solución consiste en una **descripción general de procesos**; esto es, un pro-

blema computacional tiene como solución la descripción de un conjunto de pasos que se podrían llevar a cabo de manera general para lograr un objetivo.
Un ejemplo que ilustra esto es la multiplicación. Todos sabemos multiplicar números de dos cifras, por ejemplo:

$$\begin{array}{cccc} & & 3 & 4 \\ & \times & 2 & 1 \\ \hline & & 3 & 4 \\ + & 6 & 8 & \\ \hline & 7 & 1 & 4 \end{array}$$

Pero el problema computacional asociado a la multiplicación de números de dos cifras consiste en hallar la descripción general de todos los procesos posibles de multiplicación de parejas de números de dos cifras. Este problema ha sido resuelto, desde hace varios milenios, por diferentes civilizaciones humanas, siguiendo métodos alternativos. Un método de solución moderno podría describirse así:
Tome los dos números de dos cifras, P y Q. Suponga que las cifras de P son p_1 y p_2, esto es, $P = p_1p_2$. Igualmente, suponga que $Q = q_1q_2$. La descripción **general** de todas las multiplicaciones de dos cifras puede hacerse así:

$$\begin{array}{cccc} & & p_1 & p_2 \\ & \times & q_1 & q_2 \\ \hline & & q_2p_1 & q_2p_2 \\ + & q_1p_1 & q_1p_2 & \\ \hline & q_1p_1 & q_2p_1 + q_1p_2 & q_2p_2 \end{array}$$

- Tome la cifra q_2 y multiplíquela por las cifras de P (con ayuda de una tabla de multiplicación). Ubique los resultados debajo de cada cifra de P correspondiente.

- Tome la cifra q_1 y multiplíquela por las cifras de P (con ayuda de una tabla de multiplicación). Ubique los resultados debajo de las cifras que se generaron en el paso anterior, aunque desplazadas una columna hacia la izquierda.

- Si en alguno de los pasos anteriores el resultado llega a 10 o se pasa de 10, ubique las unidades únicamente y lleve un acarreo, en decenas o centenas) para la columna de la izquierda.

- Sume los dos resultados parciales, obteniendo el resultado final.

Usted puede estar quejándose en este momento, ¿para qué hay que complicar tanto nuestro viejo y conocido proceso de multiplicación? Bueno, hay varias razones para esto:

- Una descripción impersonal como ésta puede ser leída y ejecutada por cualquier persona —o computador, como veremos mas adelante—.
- Sólo creando descripciones generales de procesos se pueden analizar para demostrar que funcionan correctamente.
- Queríamos sorprenderlo, tomando algo tan conocido como la suma y dándole una presentación que, quizás, nunca había visto. Este cambio de perspectiva es una invitación a que abra su mente a pensar en descripciones generales de procesos.

Precisando un poco, un problema computacional es la descripción general de una situación en la que se presentan unos datos de entrada y una salida deseada que se quiere calcular. Por ejemplo, en el problema computacional de la multiplicación de números de dos cifras, los datos de entrada son los números a multiplicar; la salida es el producto de los dos números. Existen más problemas computacionales como el de ordenar un conjunto de números y el problema de encontrar una palabra en un párrafo de texto. Como ejercicio defina para estos problemas cuales son los datos de entrada y la salida deseada.
La solución de un problema computacional es una descripción general del conjunto de pasos que se deben llevar a cabo con las entradas del problema para producir los datos de salida deseados. Solucionar problemas computacionales no es muy diferente de solucionar acertijos, las dos actividades producen la misma clase de retos intelectuales, y el método de solución de la sección 1.2 es aplicable en los dos casos. Lo único que hay que tener en cuenta es que la solución de un problema es una **descripción general o programa, como veremos más adelante**, que se refiere a las entradas y salidas de una manera más técnica de lo que estamos acostumbrados. Un ejemplo de esto lo constituyen los nombres p_1, p_2, q_1 y q_2 que usamos en la descripción general de la multiplicación de números de dos cifras.
Aunque la solución de problemas es una actividad compleja, es muy interesante, estimulante e intelectualmente gratificante; incluso cuando no llegamos a solucionar los problemas completamente. En el libro vamos a enfocarnos en la solución de problemas computacionales por medio de programas, y, aunque solo vamos a explorar este tipo de problemas, usted verá que las estrategias de solución, los conceptos que aprenderá y la actitud de científico de la computación que adquirirá serán valiosas herramientas para resolver todo tipo de problemas de la vida real.

1.6. Problemas computacionales propuestos

Intente resolver los siguientes problemas computacionales, proponiendo soluciones **generales e impersonales**:

- Describa cómo ordenar tres números a, b y c.
- Describa cómo encontrar el menor elemento en un conjunto de números.
- Describa cómo encontrar una palabra dentro de un texto más largo.

1.7. Glosario

Acertijo: enigma o adivinanza que se propone como pasatiempo.

Solución de problemas: el proceso de formular un problema, hallar la solución y expresar la solución.

Método de solución de problemas: un conjunto de pasos, estrategias y técnicas organizados que permiten solucionar problemas de una manera ordenada.

Problema: una situación o circunstancia en la que se dificulta lograr un fin.

Problema computacional: una situación general con una especificación de los datos de entrada y los datos de salida deseados.

Solución a un problema: conjunto de pasos y estrategias que permiten lograr un fin determinado en una situación problemática, cumpliendo ciertas restricciones.

Solución a un problema computacional: descripción general de los pasos que toman cualquier entrada en un problema computacional y la transforman en una salida deseada.

Restricción: una condición que tiene que cumplirse en un problema dado.

Capítulo 2

El camino hacia el programa

El objetivo de este libro es el de enseñar al estudiante a pensar como lo hacen los científicos informáticos. Esta manera de pensar combina las mejores características de la matemática, la ingeniería y las ciencias naturales. Como los matemáticos, los científicos informáticos usan lenguajes formales para diseñar ideas (específicamente, cómputos). Como los ingenieros, ellos diseñan cosas, construyendo sistemas mediante el ensamble de componentes y evaluando las ventajas y desventajas de cada una de las alternativas de construcción. Como los científicos, ellos observan el comportamiento de sistemas complejos, forman hipótesis, y prueban sus predicciones.

La habilidad más importante del científico informático es **la solución de problemas**. La solución de problemas incluye poder formular problemas, pensar en soluciones de manera creativa y expresar soluciones clara y precisamente. Como se verá, el proceso de aprender a programar es la oportunidad perfecta para desarrollar la habilidad de resolver problemas. Por esa razón este capítulo se llama "El camino hacia el programa".

A cierto nivel, usted aprenderá a programar, lo cual es una habilidad muy útil por sí misma. A otro nivel, usted utilizará la programación para obtener algún resultado. Ese resultado se verá más claramente durante el proceso.

2.1. El lenguaje de programación Python

El lenguaje de programación que aprenderá es Python. Python es un ejemplo de **lenguaje de alto nivel**; otros ejemplos de lenguajes de alto nivel son C, C++, Perl y Java.

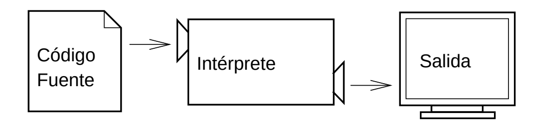
Como se puede deducir de la nomenclatura "lenguaje de alto nivel," también existen **lenguajes de bajo nivel**, que también se denominan lenguajes de máquina

o lenguajes ensambladores. A propósito, las computadoras sólo ejecutan programas escritos en lenguajes de bajo nivel. Los programas de alto nivel tienen que ser traducidos antes de ser ejecutados. Esta traducción lleva tiempo, lo cual es una pequeña desventaja de los lenguajes de alto nivel.

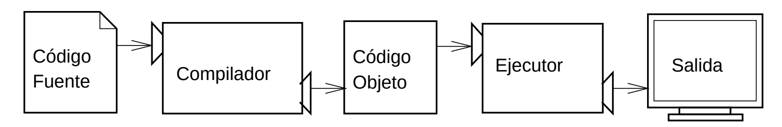
Aun así, las ventajas son enormes. En primer lugar, la programación en lenguajes de alto nivel es mucho más fácil; escribir programas en un lenguaje de alto nivel toma menos tiempo ya que los programas son más cortos, más fáciles de leer, y es más probable que queden correctos. En segundo lugar, los lenguajes de alto nivel son **portables**, lo que significa que los programas escritos con estos pueden ser ejecutados en tipos diferentes de computadoras sin modificación alguna o con pocas modificaciones. Programas escritos en lenguajes de bajo nivel sólo pueden ser ejecutados en un tipo de computadora y deben ser reescritos para ser ejecutados en otra.

Debido a estas ventajas, casi todo programa se escribe en un lenguaje de alto nivel. Los lenguajes de bajo nivel son sólo usados para unas pocas aplicaciones especiales.

Hay dos tipos de programas que traducen lenguajes de alto nivel a lenguajes de bajo nivel: **intérpretes** y **compiladores**. Una intérprete lee un programa de alto nivel y lo ejecuta, lo que significa que lleva a cabo lo que indica el programa. Traduce el programa poco a poco, leyendo y ejecutando cada comando.



Un compilador lee el programa y lo traduce todo al mismo tiempo, antes de ejecutar alguno de los programas. A menudo se compila un programa como un paso aparte, y luego se ejecuta el código compilado. En este caso, al programa de alto nivel se lo llama el **código fuente**, y al programa traducido es llamado el **código de objeto** o el **código ejecutable**.



A Python se lo considera un lenguaje interpretado, porque sus programas son ejecutados por un intérprete. Existen dos maneras de usar el intérprete: modo de comando y modo de guión. En modo de comando se escriben sentencias en el lenguaje Python y el intérprete muestra el resultado.

```
$ python
Python 2.6.2 (r262:71600, Apr 30 2009, 20:29:58)
```

```
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>1 + 1
2
```

La primera línea de este ejemplo es el comando que pone en marcha al intérprete de Python. Las dos líneas siguientes son mensajes del intérprete. La tercera línea comienza con `>>>`, lo que indica que el intérprete está listo para recibir comandos. Escribimos `1+1` y el intérprete contestó `2`.
Alternativamente, se puede escribir el programa en un archivo y usar el intérprete para ejecutar el contenido de dicho archivo. El archivo, en este caso, se denomina un **guión (script)**; por ejemplo, en un editor de texto se puede crear un archivo `unomasuno.py` que contenga esta línea:

```
print 1 + 1
```

Por acuerdo unánime, los archivos que contienen programas de Python tienen nombres que terminan con `.py`.
Para ejecutar el programa, se le tiene que indicar el nombre del guión a la interpretadora.

```
$ python unomasuno.py
2
```

En otros entornos de desarrollo, los detalles de la ejecución de programas diferirán. Además, la mayoría de programas son más interesantes que el anterior.
La mayoría de ejemplos en este libro son ejecutados en la línea de comandos. La línea de comandos es más conveniente para el desarrollo de programas y para pruebas rápidas, porque las instrucciones de Python se pueden pasar a la máquina para ser ejecutadas inmediatamente. Una vez que el programa está completo, se lo puede archivar en un guión para ejecutarlo o modificarlo en el futuro.

2.2. ¿Qué es un programa?

Un programa es una secuencia de instrucciones que especifican cómo ejecutar un cómputo. El cómputo puede ser matemático, cómo solucionar un sistema de ecuaciones o determinar las raíces de un polinomio, pero también puede ser simbólico, cómo buscar y reemplazar el texto de un documento o (aunque parezca raro) compilar un programa.
Las instrucciones (comandos, órdenes) tienen una apariencia diferente en lenguajes de programación diferentes, pero existen algunas funciones básicas que se presentan en casi todo lenguaje:

Entrada: recibir datos del teclado, o de un archivo o de otro aparato.

Salida: mostrar datos en el monitor o enviar datos a un archivo u otro aparato.

Matemáticas: ejecutar operaciones básicas, como la adición y la multiplicación.

Operación condicional: probar la veracidad de alguna condición y ejecutar una secuencia de instrucciones apropiada.

Repetición ejecutar alguna acción repetidas veces, usualmente con alguna variación.

Aunque sea difícil de creer, todos los programas en existencia son formulados exclusivamente con tales instrucciones. Así, una manera de describir la programación es: el proceso de romper una tarea en tareas cada vez más pequeñas hasta que éstas sean lo suficientemente sencillas como para ser ejecutadas con una secuencia de estas instrucciones básicas.
Quizás esta descripción es un poco ambigua. No se preocupe. Explicaremos ésto con más detalle en el tema de **algoritmos**.

2.3. ¿Qué es la depuración (debugging)?

La programación es un proceso complejo, y a veces este proceso lleva a **errores indefinidos**, también llamados **defectos** o **errores de programación** (en inglés 'bugs'), y el proceso de buscarlos y corregirlos es llamado **depuración** (en inglés 'debugging').
Hay tres tipos de errores que pueden ocurrir en un programa. Es muy útil distinguirlos para encontrarlos más rápido.

2.3.1. Errores sintácticos

Python sólo puede ejecutar un programa si está sintácticamente bien escrito. Al contrario, es decir, si el programa tiene algún error de sintaxis, el proceso falla y devuelve un mensaje de error. La palabra **sintáctica** se refiere a la estructura de cualquier programa y a las reglas de esa estructura. Por ejemplo, en español, la primera letra de toda oración debe ser mayúscula y el fin de toda oración debe llevar un punto. esta oración tiene un error sintáctico. Esta oración también
Para la mayoría de lectores, unos pocos errores no impiden la comprensión de los grafitis en la calle que, a menudo, rompen muchas reglas de sintaxis. Sin embargo Python no es así. Si hay aunque sea un error sintáctico en el programa, Python mostrará un mensaje de error y abortará su ejecución. Al principio usted pasará mucho tiempo buscando errores sintácticos, pero con el tiempo no cometerá tantos errores y los encontrará rápidamente.

2.3.2. Errores en tiempo de ejecución

El segundo tipo de error es el de tiempo de ejecución. Este error aparece sólo cuando se ejecuta un programa. Estos errores también se llaman **excepciones**, porque indican que algo excepcional ha ocurrido.
Con los programas que vamos a escribir al principio, los errores de tiempo de ejecución ocurrirán con poca frecuencia.

2.3.3. Errores semánticos

El tercer tipo de error es el **semántico**. Si hay un error de lógica en su programa, éste será ejecutado sin ningún mensaje de error, pero el resultado no será el deseado. El programa ejecutará la lógica que usted le dijo que ejecutara.
A veces ocurre que el programa escrito no es el que se tenía en mente. El sentido o significado del programa no es correcto. Es difícil hallar errores de lógica. Eso requiere trabajar al revés, comenzando a analizar la salida para encontrar al problema.

2.3.4. Depuración experimental

Una de las técnicas más importantes que usted aprenderá es la depuración. Aunque a veces es frustrante, la depuración es una de las partes de la programación más estimulantes, interesantes e intelectualmente exigentes.
La depuración es una actividad parecida a la tarea de un investigador: se tienen que estudiar las pistas para inferir los procesos y eventos que han generado los resultados encontrados.
La depuración también es una ciencia experimental. Una vez que se tiene conciencia de un error, se modifica el programa y se intenta nuevamente. Si la hipótesis fue la correcta se pueden predecir los resultados de la modificación y estaremos más cerca a un programa correcto. Si la hipótesis fue la errónea tendrá que idearse otra hipótesis. Como dijo Sherlock Holmes: "Cuando se ha descartado lo imposible, lo que queda, no importa cuan inverosímil, debe ser la verdad" (A. Conan Doyle, *The Sign of Four*)
Para algunas personas, la programación y la depuración son lo mismo: la programación es el proceso de depurar un programa gradualmente, hasta que el programa tenga el resultado deseado. Esto quiere decir que el programa debe ser, desde un principio, un programa que funcione, aunque su función sea solo mínima. El programa es depurado mientras crece y se desarrolla.
Por ejemplo, aunque el sistema operativo Linux contenga miles de líneas de instrucciones, Linus Torvalds lo comenzó como un programa para explorar el microprocesador Intel 80836. Según Larry Greenfield: "Uno de los proyectos tempranos

de Linus fue un programa que intercambiaría la impresión de AAAA con BBBB. Este programa se convirtió en Linux" (de *The Linux Users' Guide* Versión Beta 1).
Otros capítulos tratarán más el tema de la depuración y otras técnicas de programación.

2.4. Lenguajes formales y lenguajes naturales

Los **lenguajes naturales** son los hablados por seres humanos, como el español, el inglés y el francés. Éstos no han sido diseñados artificialmente (aunque se trate de imponer cierto orden en ellos), pues se han desarrollado naturalmente.
Los **Lenguajes formales** son diseñados por humanos y tienen aplicaciones específicas. La notación matemática, por ejemplo, es un lenguaje formal, ya que se presta a la representación de las relaciones entre números y símbolos. Los químicos utilizan un lenguaje formal para representar la estructura química de las moléculas. Es necesario notar que:

> **Los lenguajes de programación son formales y han sido desarrollados para expresar cómputos.**

Los lenguajes formales casi siempre tienen reglas sintácticas estrictas. Por ejemplo, $3 + 3 = 6$, es una expresión matemática correcta, pero $3 = +6\$$ no lo es. De la misma manera, $H_2 0$ es una nomenclatura química correcta, pero $_2Zz$ no lo es.
Existen dos clases de reglas sintácticas, en cuanto a unidades y estructura. Las unidades son los elementos básicos de un lenguaje, como lo son las palabras, los números y los elementos químicos. Por ejemplo, en `3=+6$`, `$` no es una unidad matemática aceptada. Similarmente, $_2Zz$ no es formal porque no hay ningún elemento químico con la abreviación Zz.
La segunda clase de error sintáctico está relacionado con la estructura de un elemento; mejor dicho, el orden de las unidades. La estructura de la sentencia `3=+6$` no es aceptada porque no se puede escribir el símbolo de igualdad seguido de un símbolo más. Similarmente, las fórmulas moleculares tienen que mostrar el número de subíndice después del elemento, no antes.

> *A manera de práctica, trate de producir una oración en español con estructura aceptada pero compuesta de unidades irreconocibles. Luego escriba otra oración con unidades aceptables pero con estructura no válida.*

Al leer una oración, sea en un lenguaje natural o una sentencia en un lenguaje técnico, se debe discernir la estructura de la oración. En un lenguaje natural este proceso, llamado **análisis sintáctico**, ocurre subconscientemente.
Por ejemplo cuando se escucha una oración simple como "el otro zapato se cayó", se puede distinguir el sustantivo "el otro zapato" y el predicado "se cayó". Cuando

se ha analizado la oración sintácticamente, se puede deducir el significado, o la semántica, de la oración. Si usted sabe lo que es un zapato y el significado de caer, comprenderá el significado de la oración.
Aunque existen muchas cosas en común entre los lenguajes naturales y los formales— por ejemplo las unidades, la estructura, la sintáctica y la semántica— también existen muchas diferencias.

Ambigüedad: los lenguajes naturales tienen muchísimas ambigüedades que se superan usando claves contextuales e información adicional. Los lenguajes formales son diseñados para estar completamente libres de ambigüedades o, tanto como sea posible, lo que quiere decir que cualquier sentencia tiene sólo un significado sin importar el contexto en el que se encuentra.

Redundancia: para reducir la ambigüedad y los malentendidos, los lenguajes naturales utilizan bastante redundancia. Como resultado tienen una abundancia de posibilidades para expresarse. Los lenguajes formales son menos redundantes y mas concisos.

Literalidad: los lenguajes naturales tienen muchas metáforas y frases comunes. El significado de un dicho, por ejemplo: "Estirar la pata", es diferente al significado de sus sustantivos y verbos. En este ejemplo, la oración no tiene nada que ver con una pata y significa 'morirse'. En los lenguajes formales solo existe el significado literal.

Los que aprenden a hablar un lenguaje natural—es decir todo el mundo—muchas veces tienen dificultad en adaptarse a los lenguajes formales. A veces la diferencia entre los lenguajes formales y los naturales es comparable a la diferencia entre la prosa y la poesía:

Poesía: se utiliza una palabra por su cualidad auditiva tanto como por su significado. El poema, en su totalidad, produce un efecto o reacción emocional. La ambigüedad no es sólo común, sino utilizada a propósito.

Prosa: el significado literal de la palabra es más importante y la estructura contribuye más al significado. La prosa se presta más al análisis que la poesía, pero todavía contiene ambigüedad.

Programa: el significado de un programa es inequívoco y literal, y es entendido en su totalidad analizando las unidades y la estructura.

He aquí unas sugerencias para la lectura de un programa (y de otros lenguajes formales). Primero, recuerde que los lenguajes formales son mucho más densos que los lenguajes naturales y, por consecuencia, toma mas tiempo dominarlos. Además, la estructura es muy importante, entonces no es una buena idea leerlo

de pies a cabeza, de izquierda a derecha. En lugar de ésto, aprenda a separar las diferentes partes en su mente, identificar las unidades e interpretar la estructura. Finalmente, ponga atención a los detalles. La fallas de puntuación y la ortografía afectarán negativamente la ejecución de sus programas.

2.5. El primer programa

Tradicionalmente el primer programa en un lenguaje nuevo se llama "Hola todo el mundo!" (en inglés, Hello world!) porque sólo muestra las palabras "Hola todo el mundo" . En el lenguaje Python es así:

```
print "Hola todo el mundo!"
```

Este es un ejemplo de una sentencia *print*, la cual no imprime nada en papel, más bien muestra un valor. En este caso, el resultado es mostrar en pantalla las palabras:

```
Hola todo el mundo!
```

Las comillas señalan el comienzo y el final del valor; no aparecen en el resultado. Hay gente que evalúa la calidad de un lenguaje de programación por la simplicidad del programa "Hola todo el mundo!". Si seguimos ese criterio, Python cumple con esta meta.

2.6. Glosario

Solución de problemas: el proceso de formular un problema, hallar la solución y expresarla.

Lenguaje de alto nivel: un lenguaje como Python que es diseñado para ser fácil de leer y escribir por la gente.

Lenguaje de bajo nivel: un lenguaje de programación que es diseñado para ser fácil de ejecutar para una computadora; también se lo llama "lenguaje de máquina" o "lenguaje ensamblador".

Portabilidad: la cualidad de un programa que puede ser ejecutado en más de un tipo de computadora.

Interpretar: ejecutar un programa escrito en un lenguaje de alto nivel traduciéndolo línea por línea.

Compilar: traducir un programa escrito en un lenguaje de alto nivel a un lenguaje de bajo nivel de una vez, en preparación para la ejecución posterior.

Código fuente: un programa escrito en un lenguaje de alto nivel antes de ser compilado.

Código objeto: la salida del compilador una vez que el programa ha sido traducido.

Programa ejecutable: otro nombre para el código de objeto que está listo para ser ejecutado.

Guión (script): un programa archivado (que va a ser interpretado).

Programa: un grupo de instrucciones que especifica un cómputo.

Algoritmo: un proceso general para resolver una clase completa de problemas.

Error (bug): un error en un programa.

Depuración: el proceso de hallazgo y eliminación de los tres tipos de errores de programación.

Sintaxis: la estructura de un programa.

Error sintáctico: un error estructural que hace que un programa sea imposible de analizar sintácticamente (e imposible de interpretar).

Error en tiempo de ejecución: un error que no ocurre hasta que el programa ha comenzado a ejecutar e impide que el programa continúe.

Excepción: otro nombre para un error en tiempo de ejecución.

Error semántico: un error en un programa que hace que ejecute algo que no era lo deseado.

Semántica: el significado de un programa.

Lenguaje natural: cualquier lenguaje hablado que evolucionó de forma natural.

Lenguaje formal: cualquier lenguaje diseñado que tiene un propósito específico, como la representación de ideas matemáticas o programas de computadoras; todos los lenguajes de programación son lenguajes formales.

Unidad: uno de los elementos básicos de la estructura sintáctica de un programa, análogo a una palabra en un lenguaje natural.

Análisis sintáctico: la revisión de un programa y el análisis de su estructura sintáctica.

Sentencia print: una instrucción que causa que el intérprete de Python muestre un valor en el monitor.

Capítulo 3

Variables, expresiones y sentencias

3.1. Valores y tipos

Un **valor** es una de las cosas fundamentales—como una letra o un número—que una progama manipula. Los valores que hemos visto hasta ahorra son `2` (el resultado cuando añadimos `1 + 1`, y `"Hola todo el Mundo!"`.

Los valores pertenecen a diferentes **tipos**: `2` es un entero, y `"Hola, Mundo!"` es una **cadena**, llamada así porque contiene una "cadena" de letras. Usted (y el intérprete) pueden identificar cadenas porque están encerradas entre comillas.

La sentencia de impresión también trabaja con enteros.

```
>>> print 4
4
```

Si no está seguro del tipo que un valor tiene, el intérprete le puede decir.

```
>>> type("Hola, Mundo!")
<type 'string'>
>>> type(17)
<type 'int'>
```

Sin despertar ninguna sorpresa, las cadenas pertenecen al tipo `string (cadena)` y los enteros pertenecen al tipo `int`. Menos obvio, los números con cifras decimales pertenecen a un tipo llamado `float`, porque éstos se representan en un formato denominado **punto flotante**.

```
>>> type(3.2)
<type 'float'>
```

¿Qué ocurre con valores como `"17"` y `"3.2"`? Parecen números, pero están encerrados entre comillas como las cadenas.

```
>>> type("17")
<type 'string'>
>>> type("3.2")
<type 'string'>
```

Ellos son cadenas.
Cuando usted digita un número grande, podría estar tentado a usar comas para separar grupos de tres dígitos, como en `1,000,000`. Esto no es un número entero legal en Python, pero esto si es legal:

```
>>> print 1,000,000
1 0 0
```

¡Bueno, eso no es lo que esperábamos!. Resulta que `1,000,000` es una tupla, algo que encontraremos en el Capítulo 10. De momento, recuerde no poner comas en sus números enteros.

3.2. Variables

Una de las características más poderosas en un lenguaje de programación es la capacidad de manipular **variables**. Una variable es un nombre que se refiere a un valor.
La **sentencia de asignación** crea nuevas variables y les da valores:

```
>>> mensaje = "¿Qué Onda?"
>>> n = 17
>>> pi = 3.14159
```

Este ejemplo hace tres asignaciones: la primera asigna la cadena `"¿Qué Onda?"` a una nueva variable denominada `mensaje`, la segunda le asigna el entero `17` a `n` y la tercera le asigna el número de punto flotante `3.14159` a `pi`.
Una manera común de representar variables en el papel es escribir el nombre de la variable con una flecha apuntando a su valor. Esta clase de dibujo se denomina **diagrama de estados** porque muestra el estado de cada una de las variables (piense en los valores como el estado mental de las variables). Este diagrama muestra el resultado de las sentencias de asignación anteriores:

mensaje → "¿Qué onda?"
n → 17
pi → 3.14159

La sentencia `print` también funciona con variables.

```
>>> print mensaje
Que Onda?
>>> print n
17
>>> print pi
3.14159
```

En cada caso el resultado es el valor de la variable. Las variables también tienen tipos; nuevamente, le podemos preguntar al intérprete cuales son.

```
>>> type(mensaje)
<type 'string'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

El tipo de una variable es el mismo del valor al que se refiere.

3.3. Nombres de variables y palabras reservadas

Los programadores, generalmente, escogen nombres significativos para sus variables —que especifiquen para qué se usa la variable.
Estos nombres pueden ser arbitrariamente largos. Pueden contener letras y números, pero tienen que empezar con una letra. Aunque es legal usar letras mayúsculas, por convención no lo hacemos. Si usted lo hace, recuerde que la capitalización importa, `Pedro` y `pedro` son variables diferentes.
El carácter subrayado (_) puede aparecer en un nombre. A menudo se usa en nombres con múltiples palabras, tales como `mi_nombre` ó `precio_del_café_en_china`.
Si usted le da un nombre ilegal a una variable obtendrá un error sintáctico:

```
>>> 76trombones = "gran desfile"
SyntaxError: invalid syntax
```

```
>>> mas$ = 1000000
SyntaxError: invalid syntax
>>> class = "introducción a la programación"
SyntaxError: invalid syntax
```

`76trombones` es ilegal porque no empieza con una letra.
`mas$` es ilegal porque contiene un carácter ilegal, el símbolo $.
¿Qué sucede con `class`?
Resulta que `class` es una de las **palabras reservadas (keywords)** de Python. Las palabras reservadas definen las reglas del lenguaje y su estructura, y no pueden ser usadas como nombres de variables.
Python tiene veintiocho palabras reservadas:

```
and       continue  else      for       import    not
assert    def       except    from      in        or
break     del       exec      global    is        pass
class     elif      finally   if        lambda    print
raise     return    try       while
```

Usted puede mantener esta lista a mano. Si el intérprete se queja por alguno de sus nombres de variables, y usted no sabe por qué, búsquelo en esta lista.

3.4. Sentencias

Una sentencia es una instrucción que el intérprete de Python puede ejecutar. Hemos visto dos clases de sentencias: la asignación y print.
Cuando usted digita una sentencia en la línea de comandos, Python la ejecuta y despliega el resultado, si hay alguno. El resultado de un print es un valor. Las asignaciones no producen un resultado.
Un guión usualmente contiene una secuencia de sentencias. Si hay más de una, los resultados aparecen uno a uno a medida que las sentencias se ejecutan.
Por ejemplo, el guión

```
print 1
x = 2
print x
```

produce la salida

```
1
2
```

Observe nuevamente que la sentencia de asignación no produce salida.

3.5. Evaluando expresiones

Una expresión es una combinación de valores, variables y operadores. Si usted digita una expresión en la línea de comandos, el intérprete la **evalúa** y despliega su resultado:

```
>>> 1 + 1
2
```

Un valor, por si mismo, se considera como una expresión, lo mismo ocurre para las variables.

```
>>> 17
17
>>> x
2
```

Aunque es un poco confuso, evaluar una expresión no es lo mismo que imprimir o desplegar un valor.

```
>>> mensaje = "Como le va, Doc?"
>>> mensaje
"Como le va, Doc?"
>>> print mensaje
Como le va, Doc?
```

Cuando Python muestra el valor de una expresión que ha evaluado, utiliza el mismo formato que se usaría para entrar un valor. En el caso de las cadenas, esto implica que se incluyen las comillas. Cuando se usa la sentencia print, el efecto es distinto como usted ya lo ha evidenciado.
En un guión, una expresión, por sí misma, es una sentencia legal, pero no realiza nada. El guión:

```
17
3.2
"Hola, Mundo!"
1 + 1
```

no produce ninguna salida. ¿Cómo cambiaría el guión de manera que despliegue los valores de las cuatro expresiones?

3.6. Operadores y operandos

Los **operadores** son símbolos especiales que representan cómputos, como la suma y la multiplicación. Los valores que el operador usa se denominan **operandos**.
Los siguientes son expresiones válidas en Python, cuyo significado es más o menos claro:

```
20+32       hora-1   hora*60+minuto
minuto/60   5**2     (5+9)*(15-7)
```

Los símbolos `+`, `-`, y `/`, y los paréntesis para agrupar, significan en Python lo mismo que en la matemática. El asterisco (`*`) es el símbolo para la multiplicación, y `**` es el símbolo para la exponenciación.
Cuando el nombre de una variable aparece en lugar de un operando, se reemplaza por su valor antes de calcular la operación
La suma, resta, multiplicación y exponenciación realizan lo que usted esperaría, pero la división podría sorprenderlo. La siguiente operación tiene un resultado inesperado:

```
>>> minuto = 59
>>> minuto/60
0
```

El valor de `minuto` es 59, y 59 dividido por 60 es 0.98333, no 0. La razón para esta discrepancia radica en que Python está realizando **división entera**.
Cuando los dos operandos son enteros el resultado también debe ser un entero; y, por convención, la división entera siempre redondea *hacia abajo*, incluso en casos donde el siguiente entero está muy cerca.
Una solución posible a este problema consiste en calcular un porcentaje, en lugar de una fracción:

```
>>> minuto*100/60
98
```

De nuevo, el resultado se redondea; pero, al menos ahora, el resultado estará mas aproximado. Otra alternativa es usar la división en punto flotante, lo que haremos en el Capítulo 4.

3.7. Orden de las operaciones

Cuando hay más de un operador en una expresión, el orden de evaluación depende de las **reglas de precedencia**. Python sigue las mismas reglas de precedencia a las que estamos acostumbrados para sus operadores matemáticos. El acrónimo **PEMDAS** es útil para recordar el orden de las operaciones:

- Los **P**aréntesis tienen la precedencia más alta y pueden usarse para forzar la evaluación de una expresión de la manera que usted desee. Ya que las expresiones en paréntesis se evalúan primero, `2 * (3-1)` es 4, y `(1+1)**(5-2)` es 8. Usted también puede usar paréntesis para que una expresión quede más legible, como en `(minuto * 100) / 60`, aunque esto no cambie el resultado.
- La **E**xponenciación tiene la siguiente precedencia más alta, así que `2**1+1` es 3 y no 4, y `3*1**3` es 3 y no 27.
- La **M**ultiplicación y la **D**ivisión tienen la misma precedencia, aunque es más alta que la de la **A**dición y la **S**ubtracción, que también tienen la misma precedencia. Así que `2*3-1` da 5 en lugar de 4, y `2/3-1` es `-1`, no `1` (recuerde que en división entera, `2/3=0`).
- Los operadores con la misma precedencia se evalúan de izquierda a derecha. Recordando que `minuto=59`, en la expresión `minuto*100/60`; la multiplicación se hace primero, resultando `5900/60`, lo que a su vez da `98`. Si las operaciones se hubieran evaluado de derecha a izquierda, el resultado sería `59/1`, que es `59`, y no es lo correcto.

3.8. Operaciones sobre cadenas

En general, usted no puede calcular operaciones matemáticas sobre cadenas, incluso si las cadenas lucen como números. Las siguientes operaciones son ilegales (asumiendo que `mensaje` tiene el tipo `cadena`):

```
mensaje-1   "Hola"/123   mensaje*"Hola"   "15"+2
```

Sin embargo, el operador `+` funciona con cadenas, aunque no calcula lo que usted esperaría. Para las cadenas, el operador `+` representa la **concatenación**, que significa unir los dos operandos enlazándolos en el orden en que aparecen. Por ejemplo:

```
fruta = "banano"
bienCocinada = " pan con nueces"
print fruta + bienCocinada
```

La salida de este programa es `banano pan con nueces`. El espacio antes de la palabra `pan` es parte de la cadena y sirve para producir el espacio entre las cadenas concatenadas.
El operador `*` también funciona con las cadenas; hace una repetición. Por ejemplo, `'Fun'*3` es `'FunFunFun'`. Uno de los operandos tiene que ser una cadena, el otro tiene que ser un entero.

Estas interpretaciones de `+` y `*` tienen sentido por la analogía con la suma y la multiplicación. Así como `4*3` es equivalente a `4+4+4`, esperamos que `"Fun"*3` sea lo mismo que `"Fun"+"Fun"+"Fun"`, y lo és. Sin embargo, las operaciones de concatenación y repetición sobre cadenas tienen una diferencia significativa con las operaciones de suma y multiplicación. ¿Puede usted pensar en una propiedad que la suma y la multiplicación tengan y que la concatenación y repetición no?

3.9. Composición

Hasta aquí hemos considerado a los elementos de un programa—variables, expresiones y sentencias—aisladamente, sin especificar cómo combinarlos.

Una de las características mas útiles de los lenguajes de programación es su capacidad de tomar pequeños bloques para **componer** con ellos. Por ejemplo, ya que sabemos cómo sumar números y cómo imprimirlos; podemos hacer las dos cosas al mismo tiempo:

```
>>>  print 17 + 3
20
```

De hecho, la suma tiene que calcularse antes que la impresión, así que las acciones no están ocurriendo realmente al mismo tiempo. El punto es que cualquier expresión que tenga números, cadenas y variables puede ser usada en una sentencia de impresión (`print`). Usted ha visto un ejemplo de esto:

```
print "Número de minutos desde media noche: ",
      hora*60+minuto
```

Usted también puede poner expresiones arbitrarias en el lado derecho de una sentencia de asignación:

```
porcentaje = (minuto * 100) / 60
```

Esto no parece nada impresionante ahora, pero vamos a ver otros ejemplos en los que la composición hace posible expresar cálculos complejos organizada y concisamente.

Advertencia: hay restricciones sobre los lugares en los que se pueden usar las expresiones. Por ejemplo, el lado izquierdo de una asignación tiene que ser un nombre de *variable*, no una expresión. Así que esto es ilegal: `minuto+1 = hora`.

3.10. Comentarios

A medida que los programas se hacen más grandes y complejos, se vuelven más difíciles de leer. Los lenguajes formales son densos; y, a menudo, es difícil mirar una sección de código y saber qué hace, o por qué lo hace.

Por esta razón, es una muy buena idea añadir notas a sus programas para explicar, en lenguaje natural, lo que hacen. Estas notas se denominan **comentarios** y se marcan con el símbolo #:

```
# calcula el porcentaje de la hora que ha pasado
porcentaje = (minuto * 100) / 60
```

En este caso, el comentario aparece en una línea completa. También pueden ir comentarios al final de una línea:

```
# precaucion: division entera
porcentaje = (minute * 100) / 60
```

Todo lo que sigue desde el # hasta el fin de la línea se ignora—no tiene efecto en el programa. El mensaje es para el programador que escribe el programa o para algún programador que podría usar este código en el futuro. En este caso, le recuerda al lector el sorprendente comportamiento de la división entera en Python.

3.11. Glosario

Valor: un número o una cadena (u otra cosa que se introduzca más adelante) que puede ser almacenado en una variable o calculado en una expresión.

Tipo: conjunto de valores. El tipo del valor determina cómo se puede usar en expresiones. Hasta aquí, los tipos que usted ha visto son enteros (tipo `int`), números de punto flotante (tipo `float`) y cadenas (tipo `string`).

Punto flotante: formato para representar números con parte decimal.

Variable: nombre que se refiere a un valor.

Sentencia: sección de código que representa un comando o acción. Hasta aquí las sentencias que usted ha visto son la de asignación y la de impresión.

Asignación: corresponde a la sentencia que pone un valor en una variable.

Diagrama de estados: es la representación gráfica de un conjunto de variables y los valores a los que se refieren.

Palabra reservada: es una palabra usada por el compilador para analizar sintácticamente un programa; usted no puede usar palabras reservadas como `if`, `def`, y `while` como nombres de variables.

Operador: símbolo especial que representa un simple cálculo como una suma, multiplicación o concatenación de cadenas.

Operando: uno de los valores sobre el cual actúa un operador.

Expresión: combinación de variables, operadores y valores que representa un único valor de resultado.

Evaluar: simplificar una expresión ejecutando varias operaciones a fin de retornar un valor único.

División entera: operación que divide un entero por otro y retorna un entero. La división entera retorna el número de veces que el denominador cabe en el numerador y descarta el residuo.

Reglas de precedencia: reglas que gobiernan el orden en que las expresiones que tienen múltiples operadores y operandos se evalúan.

Concatenar: unir dos operandos en el orden en que aparecen.

Composición: es la capacidad de combinar simples expresiones y sentencias dentro de sentencias y expresiones compuestas para representar cálculos complejos concisamente.

Comentario: información que se incluye en un programa para otro programador (o lector del código fuente) que no tiene efecto en la ejecución.

Capítulo 4

Funciones

4.1. Llamadas a funciones

Usted ya ha visto un ejemplo de una **llamada a función**:

```
>>> type("32")
<type 'string'>
```

El nombre de la función es `type`, y despliega el tipo de un valor o variable. El valor o variable, que se denomina el **argumento** de la función, tiene que encerrarse entre paréntesis. Es usual decir que una función "toma" un argumento y "retorna" un resultado. El resultado se denomina el **valor de retorno**.
En lugar de imprimir el valor de retorno, podemos asignarlo a una variable:

```
>>> betty = type("32")
>>> print betty
<type 'string'>
```

Otro ejemplo es la función `id` que toma un valor o una variable y retorna un entero que actúa como un identificador único:

```
>>> id(3)
134882108
>>> betty = 3
>>> id(betty)
134882108
```

Cada valor tiene un `id` que es un número único relacionado con el lugar en la memoria en el que está almacenado. El `id` de una variable es el `id` del valor al que la variable se refiere.

4.2. Conversión de tipos

Python proporciona una colección de funciones que convierten valores de un tipo a otro. La función `int` toma cualquier valor y lo convierte a un entero, si es posible, de lo contrario se queja:

```
>>> int("32")
32
>>> int("Hola")
ValueError: invalid literal for int(): Hola
```

`int` también puede convertir valores de punto flotante a enteros, pero hay que tener en cuenta que va a eliminar la parte decimal:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

La función `float` convierte enteros y cadenas a números de punto flotante:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
```

Finalmente, la función `str` convierte al tipo cadena (`string`):

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
```

Puede parecer extraño el hecho de que Python distinga el valor entero `1` del valor en punto flotante `1.0`. Pueden representar el mismo número pero tienen diferentes tipos. La razón para esto es que su representación interna en la memoria del computador es distinta.

4.3. Coerción de tipos

Ahora que podemos convertir entre tipos, tenemos otra forma de esquivar a la división entera. Retomando el ejemplo del capítulo anterior, suponga que deseamos calcular la fracción de una hora que ha transcurrido. La expresión más obvia

`minuto/60`, hace división entera, así que el resultado siempre es 0, incluso cuando han transcurrido 59 minutos.
Una solución es convertir `minuto` a punto flotante para realizar la división en punto flotante:

```
>>> minuto = 59
>>> float(minute)/60.0
0.983333333333
```

Otra alternativa es sacar provecho de las reglas de conversión automática de tipos, que se denominan **coerción de tipos**. Para los operadores matemáticos, si algún operando es un número `flotante`, el otro se convierte automáticamente a `flotante`:

```
>>> minuto = 59
>>> minuto / 60.0
0.983333333333
```

Así que haciendo el denominador flotante, forzamos a Python a realizar división en punto flotante.

4.4. Funciones matemáticas

En matemática usted probablemente ha visto funciones como el `seno` y el `logaritmo`, y ha aprendido a evaluar expresiones como `sen(pi/2)` y `log(1/x)`. Primero, se evalúa la expresión entre paréntesis (el argumento). Por ejemplo, `pi/2` es aproximadamente 1.571, y `1/x` es 0.1 (si `x` tiene el valor 10.0).
Entonces, se evalúa la función, ya sea mirando el resultado en una tabla o calculando varias operaciones. El `seno` de 1.571 es 1, y el `logaritmo` de 0.1 es -1 (asumiendo que `log` indica el logaritmo en base 10).
Este proceso puede aplicarse repetidamente para evaluar expresiones más complicadas como `log(1/sen(pi/2))`. Primero se evalúa el argumento de la función más interna, luego la función, y se continúa así.
Python tiene un módulo matemático que proporciona la mayoría de las funciones matemáticas. Un módulo es un archivo que contiene una colección de funciones relacionadas.
Antes de que podamos usar funciones de un módulo, tenemos que importarlas:

```
>>> import math
```

Para llamar a una de las funciones, tenemos que especificar el nombre del módulo y el nombre de la función, separados por un punto. Este formato se denomina **notación punto**.

```
>>> decibel = math.log10 (17.0)
>>> angulo = 1.5
>>> altura = math.sin(angulo)
```

La primera sentencia le asigna a `decibel` el logaritmo de 17, en base `10`. También hay una función llamada `log` que usa la base logarítmica `e`.
La tercera sentencia encuentra el seno del valor de la variable `angulo`. `sin` y las otras funciones trigonométricas (`cos`, `tan`, etc.) reciben sus argumentos en radianes. Para convertir de grados a radianes hay que dividir por 360 y multiplicar por `2*pi`. Por ejemplo, para encontrar el seno de 45 grados, primero calculamos el ángulo en radianes y luego tomamos el seno:

```
>>> grados = 45
>>> angulo = grados * 2 * math.pi / 360.0
>>> math.sin(angulo)
```

La constante `pi` también hace parte del módulo matemático. Si usted recuerda geometría puede verificar el resultado comparándolo con la raíz cuadrada de 2 dividida por 2:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

4.5. Composición

Así como las funciones matemáticas, las funciones de Python pueden componerse, de forma que una expresión sea parte de otra. Por ejemplo, usted puede usar cualquier expresión como argumento a una función:

```
>>> x = math.cos(angulo + pi/2)
```

Esta sentencia toma el valor de `pi`, lo divide por 2, y suma este resultado al valor de `angulo`. Después, la suma se le pasa como argumento a la función coseno (`cos`).
También se puede tomar el resultado de una función y pasarlo como argumento a otra:

```
>>> x = math.exp(math.log(10.0))
```

Esta sentencia halla el logaritmo en base `e` de 10 y luego eleva `e` a dicho resultado. El resultado se asigna a `x`.

4.6. Agregando nuevas funciones

Hasta aquí solo hemos usado las funciones que vienen con Python, pero también es posible agregar nuevas funciones. Crear nuevas funciones para resolver nuestros problemas particulares es una de las capacidades mas importantes de un lenguaje de programación de propósito general.

En el contexto de la programación, una **función** es una secuencia de sentencias que ejecuta una operación deseada y tiene un nombre. Esta operación se especifica en una **definición de función**. Las funciones que hemos usado hasta ahora ya han sido definidas para nosotros. Esto es bueno, porque nos permite usarlas sin preocuparnos de los detalles de sus definiciones.

La sintaxis para una definición de función es:

```
def NOMBRE( LISTA DE PARAMETROS ):
  SENTENCIAS
```

Usted puede inventar los nombres que desee para sus funciones con tal de que no use una palabra reservada. La lista de parámetros especifica que información, si es que la hay, se debe proporcionar a fin de usar la nueva función.

Se puede incluir cualquier número de sentencias dentro de la función, pero tienen que sangrarse o indentarse a partir de la margen izquierda. En los ejemplos de este libro usaremos un sangrado de dos espacios.

Las primeras funciones que vamos a escribir no tienen parámetros, así que la sintaxis luce así:

```
def nuevaLínea():
  print
```

Esta función se llama `nuevaLínea`. Los paréntesis vacíos indican que no tiene parámetros. Contiene solamente una sentencia, que produce como salida una línea vacía. Eso es lo que ocurre cuando se usa el comando `print` sin argumentos.

La sintaxis para llamar la nueva función es la misma que para las funciones predefinidas en Python:

```
print "Primera Línea."
nuevaLínea()
print "Segunda Línea."
```

La salida para este programa es:

```
Primera Línea.

Segunda Línea
```

Note el espacio extra entre las dos líneas. ¿Qué pasa si deseamos más espacio entre las líneas? Podemos llamar la misma función repetidamente:

```
print "Primera Línea."
nuevaLínea()
nuevaLínea()
nuevaLínea()
print "Segunda Línea."
```

O podemos escribir una nueva función llamada `tresLíneas` que imprima tres líneas:

```
def tresLíneas():
  nuevaLínea()
  nuevaLínea()
  nuevaLínea()

print "Primera Línea."
tresLíneas()
print "Segunda Línea."
```

Esta función contiene tres sentencias, y todas están sangradas por dos espacios. Como la próxima sentencia (print "Primera Línea") no está sangrada, Python la interpreta afuera de la función.
Hay que enfatizar dos hechos sobre este programa:

1. Usted puede llamar la misma función repetidamente. De hecho, es una práctica muy común y útil.

2. Usted puede llamar una función dentro de otra función; en este caso `tresLíneas` llama a `nuevaLínea`.

Hasta este punto, puede que no parezca claro porque hay que tomarse la molestia de crear todas estas funciones. De hecho, hay muchas razones, y este ejemplo muestra dos:

- Crear una nueva función le da a usted la oportunidad de nombrar un grupo de sentencias. Las funciones pueden simplificar un programa escondiendo un cálculo complejo detrás de un comando único que usa palabras en lenguaje natural, en lugar de un código arcano.

- Crear una nueva función puede recortar el tamaño de un programa eliminando el código repetitivo. Por ejemplo, una forma más corta de imprimir nueve líneas consecutivas consiste en llamar la función `tresLíneas` tres veces.

  Como ejercicio, escriba una función llamada `nueveLíneas` que use a `tresLíneas` para imprimir nueve líneas. ¿Como imprimiría veintisiete líneas?

4.7. Definiciones y uso

Uniendo los fragmentos de la sección 3.6, el programa completo luce así:

```
def nuevaLínea():
  print

def tresLíneas():
  nuevaLínea()
  nuevaLínea()
  nuevaLínea()

print "Primera Línea."
tresLíneas()
print "Segunda Línea."
```

Este programa contiene dos definiciones de funciones: `nuevaLínea` y `tresLíneas`. Las definiciones de funciones se ejecutan como las otras sentencias, pero su efecto es crear nuevas funciones. Las sentencias, dentro de la función, no se ejecutan hasta que la función sea llamada, y la definición no genera salida.
Como usted puede imaginar, se tiene que crear una función antes de ejecutarla. En otras palabras, la definición de función tiene que ejecutarse antes de llamarla por primera vez.

> *Como ejercicio, mueva las últimas tres líneas de este programa al inicio, de forma que los llamados a función aparezcan antes de las definiciones. Ejecute el programa y observe qué mensaje de error obtiene.*

> *Como otro ejercicio, comience con el programa original y mueva la definición de* `nuevaLínea` *después de la definición de* `tresLíneas`*. ¿Que pasa cuando se ejecuta este programa modificado?*

4.8. Flujo de ejecución

Con el objetivo de asegurar que una función se defina antes de su primer uso usted tiene que saber el orden en el que las sentencias se ejecutan, lo que denominamos **flujo de ejecución**.
La ejecución siempre empieza con la primera sentencia del programa. Las sentencias se ejecutan una a una, desde arriba hacia abajo.
Las definiciones de funciones no alteran el flujo de ejecución del programa, recuerde que las sentencias que están adentro de las funciones no se ejecutan hasta que éstas sean llamadas. Aunque no es muy común, usted puede definir una función

adentro de otra. En este caso, la definición interna no se ejecuta hasta que la otra función se llame.
Las llamadas a función son como un desvío en el flujo de ejecución. En lugar de continuar con la siguiente sentencia, el flujo salta a la primera línea de la función llamada, ejecuta todas las sentencias internas, y regresa para continuar donde estaba previamente.
Esto suena sencillo, hasta que tenemos en cuenta que una función puede llamar a otra. Mientras se está ejecutando una función, el programa puede ejecutar las sentencias en otra función. Pero, mientras se está ejecutando la nueva función, ¡el programa puede tener que ejecutar *otra* función!.
Afortunadamente, Python lleva la pista de donde está fielmente, así que cada vez que una función termina, el programa continúa su ejecución en el punto donde se la llamó. Cuando llega al fin del programa, la ejecución termina.
¿Cual es la moraleja de esta sórdida historia? Cuando lea un programa, no lo haga de arriba hacia abajo. En lugar de ésto, siga el flujo de ejecución.

4.9. Parámetros y argumentos

Algunas de las funciones primitivas que usted ha usado requieren argumentos, los valores que controlan el trabajo de la función. Por ejemplo, si usted quiere encontrar el seno de un número, tiene que indicar cual es el número. Así que, `sin` toma un valor numérico como argumento.
Algunas funciones toman más de un argumento. Por ejemplo `pow` (potencia) toma dos argumentos, la base y el exponente. Dentro de una función, los valores que se pasan se asignan a variables llamadas **parámetros**.
Aquí hay un ejemplo de una función definida por el programador que toma un parámetro:

```
def imprimaDoble(pedro):
  print pedro, pedro
```

Esta función toma un argumento y lo asigna a un parámetro llamado `pedro`. El valor del parámetro (en este momento no tenemos idea de lo que será) se imprime dos veces, y después, se imprime una línea vacía. El nombre `pedro` se escogió para sugerir que el nombre que se le asigna a un parámetro queda a su libertad; pero, en general, usted desea escoger algo mas ilustrativo que `pedro`.
La función `imprimaDoble` funciona para cualquier tipo que pueda imprimirse:

```
>>> imprimaDoble('Spam')
Spam Spam
>>> imprimaDoble(5)
5 5
```

```
>>> imprimaDoble(3.14159)
3.14159 3.14159
```

En el primer llamado de función el argumento es una cadena. En el segundo es un entero. En el tercero es un flotante (`float`).
Las mismas reglas de composición que se aplican a las funciones primitivas, se aplican a las definidas por el programador, así que podemos usar cualquier clase de expresión como un argumento para `imprimaDoble`:

```
>>> imprimaDoble('Spam'*4)
SpamSpamSpamSpam SpamSpamSpamSpam
>>> imprimaDoble(math.cos(math.pi))
-1.0 -1.0
```

Como de costumbre, la expresión se evalúa antes de que la función se ejecute así que `imprimaDoble` retorna `SpamSpamSpamSpam SpamSpamSpamSpam` en lugar de `'Spam'*4 'Spam'*4`.

> *Como ejercicio, escriba una llamada a `imprimaDoble` que retorne `'Spam'*4 'Spam'*4`. Pista: las cadenas pueden encerrarse en comillas sencillas o dobles, y el tipo de la comilla que no se usa puede usarse adentro como parte de la cadena.*

También podemos usar una variable como argumento:

```
>>> m = 'Oh, mundo cruel.'
>>> imprimaDoble(m)
Oh, mundo cruel. Oh, mundo cruel.
```

Observe algo muy importante, el nombre de la variable que pasamos como argumento (`m`) no tiene nada que ver con el nombre del parámetro (`pedro`). No importa como se nombraba el valor originalmente (en el lugar donde se hace el llamado); en la función `imprimaDoble`, la seguimos llamando de la misma manera `pedro`.

4.10. Las variables y los parámetros son locales

Cuando usted crea una **variable local** en una función, solamente existe dentro de ella, y no se puede usar por fuera. Por ejemplo:

```
def concatenarDoble(parte1, parte2):
  cat = parte1 + parte2
  imprimaDoble(cat)
```

Esta función toma dos argumentos, los concatena, y luego imprime el resultado dos veces. Podemos llamar a la función con dos cadenas:

```
>>> cantar1 = "Pie Jesu domine, "
>>> cantar2 = "Dona eis requiem."
>>> cocatenarDoble(cantar1, cantar2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona
eis requiem.
```

Cuando `concatenarDoble` termina, la variable `cat` se destruye. Si intentaramos imprimirla obtendríamos un error:

```
>>> print cat
NameError: cat
```

Los parámetros también son locales. Por ejemplo, afuera de la función `imprimaDoble`, no existe algo como `pedro`. Si usted intenta usarlo Python se quejará.

4.11. Diagramas de pila

Para llevar pista de los lugares en que pueden usarse las variables es útil dibujar un **diagrama de pila**. Como los diagramas de estados, los diagramas de pila muestran el valor de cada variable y además muestran a que función pertenece cada una. Cada función se representa por un **marco**. Un marco es una caja con el nombre de una función al lado y los parámetros y variables adentro. El diagrama de pila para el ejemplo anterior luce así:

__main__
cantar1 → "Pie Jesu domine,"
cantar2 → "Dona eis requiem."

concatenarDoble
parte1 → "Pie Jesu domine,"
parte2 → "Dona eis requiem."
cat → "Pie Jesu domine, Dona eis requiem."

imprimaDoble
pedro → "Pie Jesu domine, Dona eis requiem."

El orden de la pila muestra el flujo de ejecución. `imprimaDoble` fue llamada por `concatenarDoble`, y `concatenarDoble` fue llamada por `__main__`, que es un nombre especial para la función más superior (la principal, que tiene todo programa). Cuando usted crea una variable afuera de cualquier función, pertenece a `__main__`.

Cada parámetro se refiere al mismo valor que su argumento correspondiente. Así que `parte1` tiene el mismo valor que `cantar1`, `parte2` tiene el mismo valor que `cantar2`, y `pedro` tiene el mismo valor que `cat`.

Si hay un error durante una llamada de función, Python imprime el nombre de ésta, el nombre de la función que la llamó, y así sucesivamente hasta llegar a `__main__`.

Por ejemplo, si intentamos acceder a `cat` desde `imprimaDoble`, obtenemos un `error de nombre (NameError)`:

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    concatenarDoble(cantar1, cantar2)
  File "test.py", line 5, in concatenarDoble
    imprimaDoble(cat)
  File "test.py", line 9, in imprimaDoble
    print cat
NameError: cat
```

Esta lista de funciones se denomina un **trazado inverso**. Nos informa en qué archivo de programa ocurrió el error, en qué línea, y qué funciones se estaban ejecutando en ese momento. También muestra la línea de código que causó el error.

Note la similaridad entre el trazado inverso y el diagrama de pila. Esto no es una coincidencia.

4.12. Funciones con resultados

Usted ya puede haber notado que algunas de las funciones que estamos usando, como las matemáticas, entregan resultados. Otras funciones, como `nuevaLínea`, ejecutan una acción pero no entregan un resultado. Esto genera algunas preguntas:

1. ¿Qué pasa si usted llama a una función y no hace nada con el resultado (no lo asigna a una variable o no lo usa como parte de una expresión mas grande)?

2. ¿Qué pasa si usted usa una función sin un resultado como parte de una expresión, tal como `nuevaLínea() + 7`?

3. ¿Se pueden escribir funciones que entreguen resultados, o estamos limitados a funciones tan simples como `nuevaLínea` y `imprimaDoble`?

La respuesta a la tercera pregunta es afirmativa y lo lograremos en el capítulo 5.

> *Como ejercicio, responda las dos primeras preguntas intentándolas en Python. (Cuando usted se esté preguntando si algo es legal o ilegal una buena forma de averiguarlo es intentarlo en el intérprete).*

4.13. Glosario

Llamada a función: sentencia que ejecuta una función. Consiste en el nombre de la función seguido por una lista de argumentos encerrados entre paréntesis.

Argumento: valor que se le da a una función cuando se la está llamando. Este valor se le asigna al parámetro correspondiente en la función.

Valor de retorno: es el resultado de una función. Si una llamada a función se usa como una expresión, el valor de ésta es el valor de retorno de la función.

Conversión de tipo: sentencia explícita que toma un valor de un tipo y calcula el valor correspondiente de otro tipo.

Coerción de tipos: conversión de tipo que se hace automáticamente de acuerdo a las reglas de coerción del lenguaje de programación.

Módulo: archivo que contiene una colección de funciones y clases relacionadas.

Notación punto: sintaxis para llamar una función que se encuentra en otro módulo, especificando el nombre módulo seguido por un punto y el nombre de la función (sin dejar espacios intermedios).

Función: es la secuencia de sentencias que ejecuta alguna operación útil y que tiene un nombre definido. Las funciones pueden tomar o no tomar parámetros y pueden entregar o no entregar un resultado.

Definición de función: sentencia que crea una nueva función especificando su nombre, parámetros y las sentencias que ejecuta.

Flujo de ejecución: orden en el que las sentencias se ejecutan cuando un programa corre.

Parámetro: nombre usado dentro de una función para referirse al valor que se pasa como argumento.

Variable local: variable definida dentro de una función. Una variable local solo puede usarse dentro de su función.

Diagrama de pila: es la representación gráfica de una pila de funciones, sus variables, y los valores a los que se refieren.

Marco: una caja en un diagrama de pila que representa un llamado de función. Contiene las variables locales y los parámetros de la función.

Trazado inverso: lista de las funciones que se estaban ejecutando y que se imprime cuando ocurre un error en tiempo de ejecución.

Capítulo 5

Condicionales y recursión

5.1. El operador residuo

El **operador residuo** trabaja con enteros (y expresiones enteras) calculando el residuo del primer operando cuando se divide por el segundo. En Python este operador es un signo porcentaje (%). La sintaxis es la misma que para los otros operadores:

```
>>> cociente = 7 / 3
>>> print cociente
2
>>> residuo = 7 % 3
>>> print residuo
1
```

Así que 7 dividido por 3 da 2 con residuo 1.

El operador residuo resulta ser sorprendentemente útil. Por ejemplo, usted puede chequear si un número es divisible por otro —si `x%y` es cero, entonces `x` es divisible por `y`.

Usted también puede extraer el dígito o dígitos más a la derecha de un número. Por ejemplo, `x% 10` entrega el dígito más a la derecha de `x` (en base 10). Igualmente, `x% 100` entrega los dos últimos dígitos.

5.2. Expresiones booleanas

El tipo que Python provee para almacenar valores de verdad (cierto o falso) se denomina bool por el matemático británico George Bool. Él creó el Álgebra Booleana, que es la base para la aritmética que se usa en los computadores modernos.

Sólo hay dos valores booleanos: True (cierto) y False (falso). Las mayúsculas importan, ya que true y false no son valores booleanos.
El operador == compara dos valores y produce una expresión booleana:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

En la primera sentencia, los dos operandos son iguales, así que la expresión evalúa a True (cierto); en la segunda sentencia, 5 no es igual a 6, así que obtenemos False (falso).
El operador == es uno de los **operadores de comparación**; los otros son:

```
x != y               # x no es igual y
x > y                # x es mayor que y
x < y                # x es menor que y
x >= y               # x es mayor o igual a y
x <= y               # x es menor o igual a y
```

Aunque estas operaciones probablemente son familiares para usted, los símbolos en Python difieren de los matemáticos. Un error común consiste en usar un solo signo igual (=) en lugar en un doble signo igual (==). Recuerde que = es el operador para la asignación y que == es el operador para comparación. Tenga en cuenta que no existen los signos =< o =>.

5.3. Operadores lógicos

Hay tres **operadores lógicos**: `and`, `or` y `not`. La semántica (el significado) de ellos es similar a su significado en inglés. Por ejemplo, `x>0 and x<10` es cierto, sólo si `x` es mayor a cero *y* menor que 10.
`n%2 == 0 or n%3 == 0` es cierto si *alguna* de las condiciones es cierta, esto es, si el número es divisible por 2 *o* por 3.
Finalmente, el operador `not` niega una expresión booleana, así que `not(x>y)` es cierta si `(x>y)` es falsa, esto es, si `x` es menor o igual a `y`.
Formalmente, los operandos de los operadores lógicos deben ser expresiones booleanas, pero Python no es muy formal. Cualquier número diferente de cero se interpreta como "cierto."

```
>>>  x = 5
>>>  x and 1
1
```

```
>>>  y = 0
>>>  y and 1
0
```

En general, esto no se considera un buen estilo de programación. Si usted desea comparar un valor con cero, procure codificarlo explícitamente.

5.4. Ejecución condicional

A fin de escribir programas útiles, casi siempre necesitamos la capacidad de chequear condiciones y cambiar el comportamiento del programa en consecuencia. Las **sentencias condicionales** nos dan este poder. La más simple es la sentencia `if`:

```
if x > 0:
  print "x es positivo"
```

La expresión después de la sentencia `if` se denomina la **condición**. Si es cierta, la sentencia de abajo se ejecuta. Si no lo es, no pasa nada.
Como otras sentencias compuestas, la sentencia `if` comprende una cabecera y un bloque de sentencias:

```
CABECERA:
  PRIMERA SENTENCIA
  ...
  ULTIMA SENTENCIA
```

La cabecera comienza en una nueva línea y termina con dos puntos seguidos (:). Las sentencias sangradas o indentadas que vienen a continuación se denominan el **bloque**. La primera sentencia sin sangrar marca el fin del bloque. Un bloque de sentencias dentro de una sentencia compuesta también se denomina el **cuerpo** de la sentencia.
No hay límite en el número de sentencias que pueden aparecer en el cuerpo de una sentencia, pero siempre tiene que haber, al menos, una. Ocasionalmente, es útil tener un cuerpo sin sentencias (como un hueco para código que aún no se ha escrito). En ese caso se puede usar la sentencia `pass`, que no hace nada.

5.5. Ejecución alternativa

Una segunda forma de sentencia `if` es la ejecución alternativa en la que hay dos posibilidades y la condición determina cual de ellas se ejecuta. La sintaxis luce así:

```
if x%2 == 0:
  print x, "es par"
else:
  print x, "es impar"
```

Si el residuo de dividir `x` por 2 es 0, entonces sabemos que `x` es par, y el programa despliega un mensaje anunciando esto. Si la condición es falsa, la segunda sentencia se ejecuta. Como la condición, que es una expresión booleana, debe ser cierta o falsa, exactamente una de las alternativas se va a ejecutar. Estas alternativas se denominan **ramas**, porque, de hecho, son ramas en el flujo de ejecución.

Yéndonos "por las ramas", si usted necesita chequear la paridad (si un número es par o impar) a menudo, se podría "envolver" el código anterior en una función:

```
def imprimirParidad(x):
  if x%2 == 0:
    print x, "es par"
  else:
    print x, "es impar"
```

Para cualquier valor de `x`, `imprimirParidad` despliega un mensaje apropiado. Cuando se llama la función, se le puede pasar cualquier expresión entera como argumento.

```
>>> imprimirParidad(17)
>>> imprimirParidad(y+1)
```

5.6. Condicionales encadenados

Algunas veces hay más de dos posibilidades y necesitamos más de dos ramas. Una forma de expresar un cálculo así es un **condicional encadenado**:

```
if x < y:
  print x, "es menor que", y
elif x > y:
  print x, "es mayor que", y
else:
  print x, "y", y, "son iguales"
```

`elif` es una abreviatura de "else if." De nuevo, exactamente una de las ramas se ejecutará. No hay límite en el número de sentencias `elif`, pero la última rama tiene que ser una sentencia `else`:

```
if eleccion == 'A':
  funcionA()
elif eleccion == 'B':
```

```
  funcionB()
elif eleccion == 'C':
  funcionC()
else:
  print "Eleccion incorrecta."
```

Cada condición se chequea en orden. Si la primera es falsa, se chequea la siguiente, y así sucesivamente. Si una de ellas es cierta, se ejecuta la rama correspondiente y la sentencia termina. Si hay más de una condición cierta, sólo la primera rama que evalúa a cierto se ejecuta.

> *Como ejercicio, envuelva estos ejemplos en funciones llamadas* `comparar(x,y)` *y* `despachar(eleccion)`*.*

5.7. Condicionales anidados

Un condicional también se puede anidar dentro de otro. La tricotomía anterior se puede escribir así:

```
if x == y:
  print x, "y", y, "son iguales"
else:
  if x < y:
    print x, "es menor que", y
  else:
    print x, "es mayor que", y
```

El condicional externo contiene dos ramas: la primera contiene una sentencia de salida sencilla, la segunda contiene otra sentencia `if`, que tiene dos ramas propias. Esas dos ramas son sentencias de impresión, aunque también podrían ser sentencias condicionales.

Aunque la indentación o sangrado de las sentencias sugiere la estructura, los condicionales anidados rápidamente se hacen difíciles de leer. En general, es una buena idea evitarlos cada vez que se pueda.

Los operadores lógicos proporcionan formas de simplificar las sentencias condicionales anidadas. Por ejemplo, podemos reescribir el siguiente código usando un solo condicional:

```
if 0 < x:
  if x < 10:
    print "x es un digito positivo."
```

La sentencia `print` se ejecuta solamente si el flujo de ejecución ha pasado las dos condiciones, así que podemos usar el operador `and`:

```
if 0 < x and x < 10:
  print "x es un digito positivo."
```

Esta clase de condiciones es muy común, por esta razón Python proporciona una sintaxis alternativa que es similar a la notación matemática:

```
if 0 < x < 10:
  print "x es un digito positivo"
```

Desde el punto de vista semántico ésta condición es la misma que la expresión compuesta y que el condicional anidado.

5.8. La sentencia `return`

La sentencia `return` permite terminar la ejecución de una función antes de llegar al final. Una razón para usarla es reaccionar a una condición de error:

```
import math

def imprimirLogaritmo(x):
  if x <= 0:
    print "Numeros positivos solamente. Por favor"
    return

  result = math.log(x)
  print "El logaritmo de ",  x ," es ", result
```

La función `imprimirLogaritmo` toma un parámetro denominado `x`. Lo primero que hace es chequear si `x` es menor o igual a 0, caso en el que despliega un mensaje de error y luego usa a `return` para salir de la función. El flujo de ejecución inmediatamente retorna al punto donde se había llamado la función, y las líneas restantes de la función no se ejecutan.
Recuerde que para usar una función del módulo matemático (math) hay que importarlo previamente.

5.9. Recursión

Hemos mencionado que es legal que una función llame a otra, y usted ha visto varios ejemplos así. Hemos olvidado mencionar el hecho de que una función también puede llamarse a sí misma. Al principio no parece algo útil, pero resulta ser

una de las capacidades más interesantes y mágicas que un programa puede tener. Por ejemplo, observe la siguiente función:

```
def conteo(n):
  if n == 0:
    print "Despegue!"
  else:
    print n
    conteo(n-1)
```

`conteo` espera que el parámetro `n` sea un número entero positivo. Si `n` es 0, despliega la cadena, "Despegue!". Si no lo es, despliega `n` y luego llama a la función llamada `conteo`—ella misma—pasando a `n-1` como argumento.
Analizemos lo que sucede si llamamos a esta función así:

```
>>> conteo(3)
```

La ejecución de `conteo` comienza con `n=3`, y como `n` no es 0, despliega el valor 3, y se llama a sí misma ...

> La ejecución de `conteo` comienza con `n=2`, y como `n` no es 0, despliega el valor 2, y se llama a si misma ...
>
> > La ejecución de `conteo` comienza con `n=1`, y como `n` no es 0, despliega el valor 1, y se llama a sí misma ...
> >
> > > La ejecución de `conteo` comienza con `n=0`, y como `n` es 0, despliega la cadena "Despegue!" y retorna (finaliza).
> >
> > El `conteo` que recibió `n=1` retorna.
>
> El `conteo` que recibió `n=2` retorna.

El `conteo` que recibió `n=3` retorna.
Y el flujo de ejecución regresa a `__main__` (vaya viaje!). Así que, la salida total luce así:

```
3
2
1
Despegue!
```

Como otro ejemplo, utilizaremos nuevamente las funciones `nuevaLínea` y `tresLíneas`:

```
def nuevalinea():
  print

def tresLíneas():
  nuevaLínea()
  nuevaLínea()
  nuevaLínea()
```

Este trabajo no sería de mucha ayuda si quisiéramos desplegar 2 líneas o 106. Una mejor alternativa sería:

```
def nLíneas(n):
  if n > 0:
    print
    nLíneas(n-1)
```

Esta función es similar a `conteo`; en tanto `n` sea mayor a 0, despliega una nueva línea y luego se llama a sí misma para desplegar `n-1` líneas adicionales. Así, el número total de nuevas líneas es `1 + (n - 1)` que, si usted verifica con álgebra, resulta ser `n`.

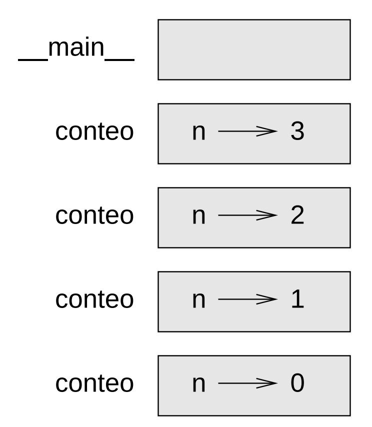
El proceso por el cual una función se llama a sí misma es la **recursión**, y se dice que estas funciones son recursivas.

5.10. Diagramas de pila para funciones recursivas

En la Sección 4.11, usamos un diagrama de pila para representar el estado de un programa durante un llamado de función. La misma clase de diagrama puede ayudarnos a interpretar una función recursiva.

Cada vez que una función se llama, Python crea un nuevo marco de función que contiene los parámetros y variables locales de ésta. Para una función recursiva, puede existir más de un marco en la pila al mismo tiempo.

Este es el diagrama de pila para `conteo` llamado con `n = 3`:



Como siempre, el tope de la pila es el marco para `__main__`. Está vacío porque no creamos ninguna variable en `__main__` ni le pasamos parámetros.
Los cuatro marcos de `conteo` tienen diferentes valores para el parámetro `n`. El fondo de la pila, donde `n=0`, se denomina el **caso base** . Como no hace una llamada recursiva, no hay mas marcos.

Como ejercicio, dibuje un diagrama de pila para `nLíneas` llamada con `n=4`.

5.11. Recursión infinita

Si una función recursiva nunca alcanza un caso base va a hacer llamados recursivos por siempre y el programa nunca termina. Esto se conoce como **recursión infinita**, y, generalmente, no se considera una buena idea. Aquí hay un programa minimalista con recursión infinita:

```
def recurrir():
  recurrir()
```

En la mayoría de ambientes de programación un programa con recursión infinita no corre realmente para siempre. Python reporta un mensaje de error cuando alcanza la máxima profundidad de recursión:

```
  File "<stdin>", line 2, in recurrir
  (98 repeticiones omitidas)
  File "<stdin>", line 2, in recurrir
RuntimeError: Maximum recursion depth exceeded
```

Este trazado inverso es un poco más grande que el que vimos en el capítulo anterior. Cuando se presenta el error, ¡hay más de 100 marcos de `recurrir` en la pila!.

Como ejercicio, escriba una función con recursión infinita y córrala en el intérprete de Python.

5.12. Entrada por el teclado

Los programas que hemos escrito son un poco toscos ya que no aceptan entrada de un usuario. Sólo hacen la misma operación todo el tiempo.

Python proporciona funciones primitivas que obtienen entrada desde el teclado. La más sencilla se llama `raw_input`. Cuando esta función se llama el programa se detiene y espera a que el usuario digite algo. Cuando el usuario digita la tecla Enter o Intro, el programa retoma la ejecución y `raw_input` retorna lo que el usuario digitó como una cadena (`string`):

```
>>> entrada = raw_input ()
Que esta esperando?
>>> print entrada
Que esta esperando?
```

Antes de llamar a `raw_input` es una muy buena idea desplegar un mensaje diciéndole al usuario qué digitar. Este mensaje se denomina indicador de entrada (**prompt** en inglés). Podemos dar un argumento prompt a `raw_input`:

```
>>> nombre = raw_input ("Cual es tu nombre? ")
Cual es tu nombre? Arturo, Rey de los Bretones!
>>> print nombre
Arturo, Rey de los Bretones!
```

Si esperamos que la respuesta sea un entero, podemos usar la función `input`:

```
prompt = "¿Cual es la velocidad de una golondrina sin carga?\n"
velocidad = input(prompt)
```

Si el usuario digita una cadena de dígitos, éstos se convierten a un entero que se asigna a `velocidad`. Desafortunadamente, si el usuario digita un carácter que no sea un dígito, el programa se aborta:

```
>>> velocidad = input (prompt)
prompt = "¿Cual es la velocidad una golondrina sin carga?\n"
¿Que quiere decir, una golondria Africana o Europea?
SyntaxError: invalid syntax
```

Para evitar este error, es una buena idea usar `raw_input` para obtener una cadena y las funciones de conversión para transformarla en otros tipos.

5.13. Glosario

Operador residuo: operador que se denota con un signo porcentaje (%), y trabaja sobre enteros produciendo el residuo de un número al dividirlo por otro.

Expresión booleana: expresión cierta o falsa.

Operador de comparación: uno de los operadores que compara dos valores: ==, !=, >, <, >=, y <=.

Operador lógico: uno de los operadores que combina expresiones booleanas: `and`, `or`, y `not`.

Sentencia condicional: sentencia que controla el flujo de ejecución dependiendo de alguna condición.

Condición: la expresión booleana en una sentencia condicional que determina que rama se ejecuta.

Sentencia compuesta: es la sentencia que comprende una cabecera y un cuerpo. La cabecera termina con dos puntos seguidos (:). El cuerpo se sangra o indenta con respecto a la cabecera.

Bloque: grupo de sentencias consecutivas con la misma indentación.

Cuerpo: el bloque, en una sentencia compuesta, que va después de la cabecera.

Anidamiento: situación en la que hay una estructura dentro de otra, tal como una sentencia condicional dentro de una rama de otra sentencia condicional.

Recursión: es el proceso de llamar la función que se está ejecutando actualmente.

Caso base: corresponde a una rama de la sentencia condicional dentro de una función recursiva, que no hace un llamado recursivo.

Recursión infinita: función que se llama a sí misma recursivamente sin alcanzar nunca el caso base. En Python una recursión infinita eventualmente causa un error en tiempo de ejecución.

Prompt (indicador de entrada): una pista visual que le indica al usuario que digite alguna información.

Capítulo 6

Funciones fructíferas

6.1. Valores de fetorno

Algunas de las funciones primitivas que hemos usado, como las matemáticas, entregan resultados. El llamar a estas funciones genera un valor nuevo, que usualmente asignamos a una variable o usamos como parte de una expresión.

```
e = math.exp(1.0)
altura = radio * math.sin(angulo)
```

Pero hasta ahora ninguna de las funciones que hemos escrito ha retornado un valor.

En este capítulo vamos a escribir funciones que retornan valores, los cuales denominamos **funciones fructíferas**, o provechosas[1]. El primer ejemplo es `area`, que retorna el área de un círculo dado su radio:

```
import math

def area(radio):
  temp = math.pi * radio**2
  return temp
```

Ya nos habíamos topado con la sentencia `return` antes, pero, en una función fructífera, la sentencia `return` incluye un **valor de retorno**. Esta sentencia significa: "Retorne inmediatamente de esta función y use la siguiente expresión como

[1]En algunos libros de programación, las *funciones* que desarrollamos en el capítulo anterior se denominan *procedimientos* y las que veremos en este capítulo sí se denominan *funciones*, ya que los lenguajes de programación usados para enseñar (como Pascal) hacían la distinción. Muchos lenguajes de programación vigentes (incluido Python y C) no diferencian sintacticamente entre procedimientos y funciones, por eso usamos esta terminología

un valor de retorno.” La expresión proporcionada puede ser arbitrariamente compleja, así que podríamos escribir esta función más concisamente:

```
def area(radio):
  return math.pi * radio**2
```

Por otro lado, las **variables temporales**, como `temp`, a menudo permiten depurar los programas más fácilmente.
Algunas veces es muy útil tener múltiples sentencias return, ubicadas en ramas distintas de un condicional:

```
def valorAbsoluto(x):
  if x < 0:
    return -x
  else:
    return x
```

Ya que estas sentencias `return` están en un condicional alternativo, sólo una será ejecutada. Tan pronto como esto suceda, la función termina sin ejecutar las sentencias que siguen.
El código que aparece después de la sentencia `return`, o en un lugar que el flujo de ejecución nunca puede alcanzar, se denomina **código muerto**.
En una función fructífera es una buena idea garantizar que toda ruta posible de ejecución del programa llegue a una sentencia `return`. Por ejemplo:

```
def valorAbsoluto(x):
  if x < 0:
    return -x
  elif x > 0:
    return x
```

Este programa no es correcto porque si `x` llega a ser 0, ninguna condición es cierta y la función puede terminar sin alcanzar una sentencia `return`. En este caso el valor de retorno que Python entrega es un valor especial denominado `None`:

```
>>> print valorAbsoluto(0)
None
```

> *Como ejercicio, escriba una función* `comparar` *que retorne* `1` *si* `x>y`*,* `0` *si* `x==y`*, y* `-1` *si* `x<y`*.*

6.2. Desarrollo de programas

En este momento usted debería ser capaz de leer funciones completas y deducir lo que hacen. También, si ha realizado los ejercicios, ya ha escrito algunas funciones pequeñas. A medida en que usted escriba funciones más grandes puede empezar a tener una dificultad mayor, especialmente con los errores semánticos y de tiempo de ejecución.

Para desarrollar programas cada vez más complejos, vamos a sugerir una técnica denominada **desarrollo incremental**. El objetivo del desarrollo incremental es evitar largas sesiones de depuración mediante la adición y prueba de una pequeña cantidad de código en cada paso.

Como ejemplo, suponga que usted desea hallar la distancia entre dos puntos dados por las coordenadas (x_1, y_1) y (x_2, y_2). Por el teorema de Pitágoras, la distancia se calcula con:

$$distancia = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{6.1}$$

El primer paso es considerar cómo luciría la función `distancia` en Python. En otras palabras, ¿cuales son las entradas (parámetros) y cual es la salida (valor de retorno)?

En este caso, los dos puntos son las entradas, que podemos representar usando cuatro parámetros. El valor de retorno es la distancia, que es un valor de punto flotante.

Ya podemos escribir un borrador de la función:

```
def distancia(x1, y1, x2, y2):
  return 0.0
```

Obviamente, esta versión de la función no calcula distancias; siempre retorna cero. Pero es correcta sintácticamente y puede correr, lo que implica que la podemos probar antes de que la hagamos más compleja.

Para probar la nueva función la llamamos con valores simples:

```
>>> distancia(1, 2, 4, 6)
0.0
```

Escogemos estos valores de forma que la distancia horizontal sea 3 y la vertical 4; de esta forma el resultado es 5 (la hipotenusa de un triángulo con medidas 3-4-5). Cuando probamos una función es fundamental conocer algunas respuestas correctas.

En este punto hemos confirmado que la función está bien sintácticamente, y que podemos empezar a agregar líneas de código. Después de cada cambio, probamos la función otra vez. Si hay un error, sabemos dónde debe estar —en la última línea que agregamos.

Un primer paso lógico en este cómputo es encontrar las diferencias $x_2 - x_1$ y $y_2 - y_1$. Almacenaremos estos valores en variables temporales llamadas `dx` y `dy` y los imprimiremos.

```
def distancia(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  print "dx es", dx
  print "dy es", dy
  return 0.0
```

Si la función trabaja bien, las salidas deben ser 3 y 4. Si es así, sabemos que la función está obteniendo los parámetros correctos y calculando el primer paso correctamente. Si no ocurre ésto, entonces hay unas pocas líneas para chequear.
Ahora calculamos la suma de los cuadrados de `dx` y `dy`:

```
def distancia(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  discuadrado = dx**2 + dy**2
  print "discuadrado es: ", discuadrado
  return 0.0
```

Note que hemos eliminado las sentencias `print` que teníamos en el paso anterior. Este código se denomina **andamiaje** porque es útil para construir el programa pero no hace parte del producto final.
De nuevo, corremos el programa y chequeamos la salida (que debe ser 25).
Finalmente, si importamos el módulo math, podemos usar la función `sqrt` para calcular y retornar el resultado:

```
def distancia(x1, y1, x2, y2):
  dx = x2 - x1
  dy = y2 - y1
  discuadrado = dx**2 + dy**2
  resultado = math.sqrt(discuadrado)
  return resultado
```

Si esto funciona bien, usted ha terminado. Si no, se podría imprimir el valor de `resultado` antes de la sentencia return.
Recapitulando, para empezar, usted debería agregar solamente una línea o dos cada vez.
A medida que gane más experiencia podrá escribir y depurar trozos mayores. De cualquier forma el proceso de desarrollo incremental puede evitarle mucho tiempo de depuración.
Los aspectos claves del proceso son:

1. Empezar con un programa correcto y hacer pequeños cambios incrementales. Si en cualquier punto hay un error, usted sabrá exactamente donde está.
2. Use variables temporales para almacenar valores intermedios de manera que se puedan imprimir y chequear.
3. Ya que el programa esté corriendo, usted puede remover parte del andamiaje o consolidar múltiples sentencias en expresiones compuestas, pero sólo si ésto no dificulta la lectura del programa.

Como ejercicio, desarrolle incrementalmente una función llamada `hipotenusa`*, que retorne la longitud de la hipotenusa de un triángulo rectángulo dadas las longitudes de los dos catetos como parámetros. Guarde cada etapa del proceso de desarrollo a medida que avanza.*

6.3. Composición

Como usted esperaría, se puede llamar una función fructífera desde otra. Esta capacidad es la **composición**.
Como ejemplo vamos a escribir una función que toma dos puntos: el centro de un círculo y un punto en el perímetro, y que calcule el área total del círculo.
Asuma que el punto central está almacenado en las variables `xc` y `yc`, y que el punto perimetral está en `xp` y `yp`. El primer paso es encontrar el radio del círculo, que es la distancia entre los dos puntos. Afortunadamente, hay una función, `distancia`, que hace eso:

```
radio = distancia(xc, yc, xp, yp)
```

El segundo paso es encontrar el área de un círculo con dicho radio y retornarla:

```
resultado = area(radio)
return resultado
```

Envolviendo todo en una función obtenemos:

```
def area2(xc, yc, xp, yp):
  radio = distancia(xc, yc, xp, yp)
  resultado = area(radio)
  return resultado
```

Llamamos a esta función `area2` para distinguirla de la función `area` definida previamente. Solo puede haber una función con un nombre dado dentro de un módulo.

Las variables temporales `radio` y `area` son útiles para desarrollar y depurar, pero una vez que el programa está funcionando podemos hacer la función más concisa componiendo las llamadas a funciones:

```
def area2(xc, yc, xp, yp):
  return area(distancia(xc, yc, xp, yp))
```

> *Como ejercicio, escriba una función* `pendiente(x1, y1, x2, y2)` *que retorna la pendiente de una línea que pasa por los puntos (x1, y1) y (x2, y2). Ahora, use esta función dentro de una función llamada* `interceptar(x1, y1, x2, y2)` *que retorna la intercepción con el eje y de la línea que pasa por los puntos* `(x1, y1)` *y* `(x2, y2)`*.*

6.4. Funciones booleanas

Las funciones que pueden retornar un valor booleano son convenientes para ocultar chequeos complicados adentro de funciones. Por ejemplo:

```
def esDivisible(x, y):
  if x % y == 0:
    return True       #  es cierto
  else:
    return False      # es falso
```

El nombre de esta función es `esDivisible`. Es muy usual nombrar las funciones booleanas con palabras o frases que suenan como preguntas de sí o no (que tienen como respuesta un sí o un no). `esDivisible` retorna `True` ó `False` para indicar si x es divisible exactamente por y.
Podemos hacerla más concisa tomando ventaja del hecho de que una condición dentro de una sentencia `if` es una expresión booleana. Podemos retornarla directamente, evitando completamente el `if`:

```
def esDivisible(x, y):
  return x % y == 0
```

Esta sesión muestra la nueva función en acción:

```
>>>   esDivisible(6, 4)
False
>>>   esDivisible(6, 3)
True
```

Las funciones booleanas se usan a menudo en las sentencias condicionales:

```
if esDivisible(x, y):
  print "x es divisible por y"
else:
  print "x no es divisible por y"
```

Puede parecer tentador escribir algo como:

```
if esDivisible(x, y) == True:
```

Pero la comparación extra es innecesaria.

> *Como ejercicio escriba una función* `estaEntre(x, y, z)` *que retorne* `True` *si* $y \leq x \leq z$ *o* `False` *si no ocurre esto.*

6.5. Más recursión

Hasta aquí, usted sólo ha aprendido un pequeño subconjunto de Python, pero podría interesarle saber que este subconjunto es un lenguaje de programación *completo*, lo que quiere decir que cualquier cosa que pueda ser calculada puede ser expresada en este subconjunto. Cualquier programa escrito alguna vez puede ser reescrito usando solamente las características que usted ha aprendido hasta ahora (de hecho, necesitaría algunos comandos mas para manejar dispositivos como el teclado, el ratón, los discos, etc., pero eso sería todo).
Demostrar esta afirmación no es un ejercicio trivial y fue logrado por Alan Turing, uno de los primeros científicos de la computación (algunos dirían que el era un matemático, pero la mayoría de los científicos pioneros de la computación eran matemáticos). Esto se conoce como la Tesis de Turing. Si usted toma un curso de Teoría de la Computación tendrá la oportunidad de ver la demostración.
Para darle una idea de lo que puede hacer con las herramientas que ha aprendido, vamos a evaluar unas pocas funciones matemáticas definidas recursivamente.
Una definición recursiva es similar a una circular, ya que éstas contienen una referencia al concepto que se pretende definir. Una definición circular verdadera no es muy útil:

frabjuoso: un adjetivo usado para describir algo que es frabjuoso.

Si usted viera dicha definición en el diccionario, quedaría confundido. Por otro lado, si encontrara la definición de la función factorial hallaría algo como esto:

$$0! = 1$$

$$n! = n(n-1)!$$

Esta definición dice que el factorial de 0 es 1, y que el factorial de cualquier otro valor, n, es n multiplicado por el factorial de $n-1$.

Así que $3!$ es 3 veces $2!$, que es 2 veces $1!$, que es 1 vez $0!$. Juntando todo esto, $3!$ es igual a 3 veces 2 veces 1 vez 1, lo que da 6.
Si usted puede escribir una definición recursiva de algo, usualmente podrá escribir un programa para evaluarlo. El primer paso es decidir cuales son los parámetros para esta función. Con un poco de esfuerzo usted concluiría que `factorial` recibe un único parámetro:

```
def factorial(n):
```

Si el argumento es 0, todo lo que hacemos es retornar 1:

```
def factorial(n):
  if n == 0:
    return 1
```

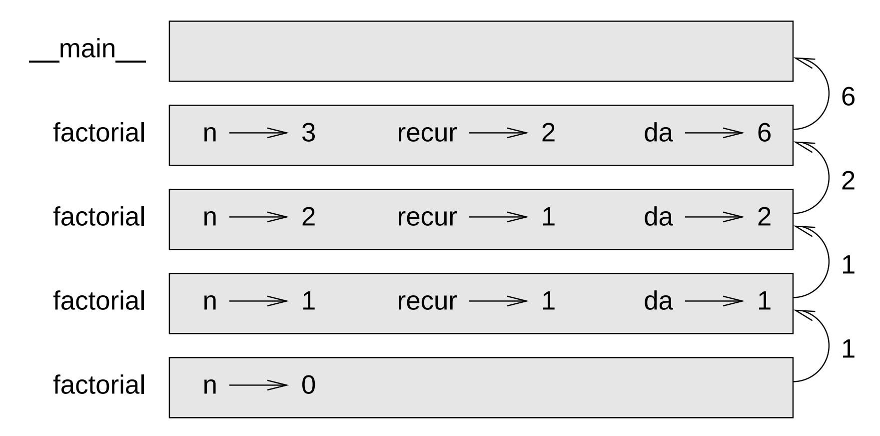
Sino, y ésta es la parte interesante, tenemos que hacer una llamada recursiva para encontrar el factorial de $n - 1$ y, entonces, multiplicarlo por n:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    recur = factorial(n-1)
    da = n * recur
    return da
```

El flujo de ejecución de este programa es similar al flujo de `conteo` en la Sección 5.9. Si llamamos a `factorial` con el valor 3:
Como 3 no es 0, tomamos la segunda rama y calculamos el factorial de `n-1`...

> Como 2 no es 0, tomamos la segunda rama y calculamos el factorial de `n-1`...
>
> > Como 1 no es 0, tomamos la segunda rama y calculamos el factorial de `n-1`...
> >
> > > Como 0 *es* 0, tomamos la primera rama y retornamos 1 sin hacer más llamados recursivos.
> >
> > El valor de retorno (1) se multiplica por n, que es 1, y el resultado se retorna.
>
> El valor de retorno (1) se multiplica por n, que es 2, y el resultado se retorna.

El valor de retorno (2) se multiplica por n, que es 3, y el resultado, 6, se convierte en el valor de retorno del llamado de función que empezó todo el proceso.
Así queda el diagrama de pila para esta secuencia de llamados de función:



Los valores de retorno mostrados se pasan hacia arriba a través de la pila. En cada marco, el valor de retorno es el valor de `da`, que es el producto de `n` y `recur`.
Observe que en el último marco, las variables locales `recur` y `da` no existen porque la rama que las crea no se ejecutó.

6.6. El salto de fe

Seguir el flujo de ejecución es una forma de leer programas, pero rápidamente puede tornarse algo laberíntico. Una alternativa es lo que denominamos hacer el "salto de fe." Cuando usted llega a un llamado de función, en lugar de seguir el flujo de ejecución, se *asume* que la función trabaja correctamente y retorna el valor apropiado.
De hecho, usted ya está haciendo el salto de fe cuando usa las funciones primitivas. Cuando llama a `math.cos` ó a `math.exp`, no está examinando las implementaciones de estas funciones. Usted sólo asume que están correctas porque los que escribieron el módulo math son buenos programadores.
Lo mismo se cumple para una de sus propias funciones. Por ejemplo, en la Sección 6.4, escribimos una función llamada `esDivisible` que determina si un número es divisible por otro. Una vez que nos hemos convencido de que esta función es correcta —probándola y examinando el código—podemos usarla sin mirar el código nuevamente.
Lo mismo vale para los programas recursivos. Cuando usted llega a una llamada recursiva, en lugar de seguir el flujo de ejecución, debería asumir que el llamado

recursivo funciona (retorna el resultado correcto) y luego preguntarse, "Asumiendo que puedo encontrar el factorial de $n - 1$, ¿puedo calcular el factorial de n?" En este caso, es claro que se puede lograr, multiplicándolo por n.
Por supuesto que es un poco raro asumir que la función trabaja correctamente cuando ni siquiera hemos terminado de escribirla, ¡por eso es que denominamos a esto el salto de fe!.

6.7. Un ejemplo más

En el ejemplo anterior usábamos variables temporales para desplegar los pasos y depurar el código más fácilmente, pero podríamos ahorrar unas cuantas líneas:

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

Desde ahora, vamos a usar esta forma más compacta, pero le recomendamos que use la forma más explícita mientras desarrolla las funciones. Cuando estén terminadas y funcionando, con un poco de inspiración se pueden compactar.
Después de `factorial`, el ejemplo más común de función matemática, definida recursivamente, es la serie de `fibonacci`, que tiene la siguiente definición:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);$$

Traducida a Python, luce así:

```
def fibonacci (n):
  if n == 0 or n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

Si usted intenta seguir el flujo de ejecución de fibonacci, incluso para valores pequeños de n, le va a doler la cabeza. Pero, si seguimos el salto de fe, si asumimos que los dos llamados recursivos funcionan correctamente, es claro que el resultado correcto es la suma de éstos dos.

6.8. Chequeo de tipos

¿Qué pasa si llamamos a `factorial` y le pasamos a 1.5 como argumento?

```
>>> factorial (1.5)
RuntimeError: Maximum recursion depth exceeded
```

Parece recursión infinita. ¿Cómo puede darse? Hay un caso base —cuando `n == 0`. El problema reside en que los valores de `n` se *saltan* al caso base .
En la primera llamada recursiva el valor de `n` es 0.5. En la siguiente es -0.5. Desde allí se hace cada vez más pequeño, pero nunca será 0.
Tenemos dos opciones, podemos intentar generalizar la función `factorial` para que trabaje con números de punto flotante, o podemos chequear el tipo del parámetro que llega. La primera opción se denomina en matemática la función gama y está fuera del alcance de este libro. Optaremos por la segunda.
Podemos usar la función `type` para comparar el tipo del parámetro al tipo de un valor entero conocido (como 1). Mientras estamos en eso también aseguraremos que el parámetro sea positivo:

```
def factorial (n):
  if type(n) != type(1):
    print "Factorial solo esta definido para enteros."
    return -1
  elif n < 0:
    print "Factorial solo esta definido para positivos"
    return -1
  elif n == 0:
    return 1
  else:
    return n * factorial(n-1)
```

Ahora tenemos tres casos base. El primero atrapa a los valores que no son enteros, el segundo a los enteros negativos. En ambos casos el programa imprime un mensaje de error y retorna un valor especial, -1, para indicar que algo falló:

```
>>> factorial ("pedro")
Factorial solo esta definido para enteros.
-1
>>> factorial (-2)
Factorial solo esta definido para positivos.
-1
```

Si pasamos los dos chequeos, tenemos la garantía de que n es un número entero positivo, y podemos probar que la recursión termina.

Este programa demuestra el uso de un patrón denominado **guarda**. Los primeros dos condicionales actúan como guardas, protegiendo al código interno de los valores que pueden causar un error. Las guardas hacen posible demostrar que el código es correcto.

6.9. Glosario

Función fructífera: función que retorna un resultado.

Valor de retorno: el valor que entrega como resultado un llamado de función.

Variable temporal: variable usada para almacenar un valor intermedio en un cálculo complejo.

Código muerto: parte de un programa que nunca puede ser ejecutada, a menudo porque aparece después de una sentencia `return`.

`None`: valor especial en Python retornado por las funciones que no tienen una sentencia return, o que tienen una sentencia return sin un argumento.

Desarrollo incremental: un plan de desarrollo de programas que evita la depuración, agregando y probando solo pequeñas porciones de código en cada momento.

Andamiaje: código que se usa durante el desarrollo de programas, pero no hace parte de la solución final.

Guarda: una condición que chequea y controla circunstancias que pueden causar errores.

Capítulo 7

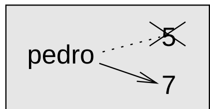
Iteración

7.1. Asignación múltiple

Puede que usted ya haya descubierto que es posible realizar más de una asignación a la misma variable. Una nueva asignación hace que la variable existente se refiera a un nuevo valor (y deje de referirse al viejo valor).

```
pedro = 5
print pedro,
pedro = 7
print pedro
```

La salida de este programa es `5 7`, porque la primera vez que `pedro` se imprime, tiene el valor 5, y la segunda vez tiene el valor 7. La coma al final del primer `print` suprime la nueva línea que tradicionalmente introduce Python después de los datos, por esta razón las dos salidas aparecen en la misma línea.
Aquí se puede ver cómo luce la **asignación múltiple** en un diagrama de estado:



Con asignación múltiple es muy importante distinguir entre una asignación y una igualdad. Como Python usa el signo igual (=) para la asignación podemos caer en la tentación de interpretar a una sentencia como `a = b` como si fuera una igualdad. ¡Y no lo es!
Primero, la igualdad es conmutativa y la asignación no lo es. Por ejemplo, en la matemática si $a = 7$ entonces $7 = a$. Pero en Python, la sentencia `a = 7` es legal aunque `7 = a` no lo es.

Además, en matemática, una sentencia de igualdad *siempre* es cierta. Si $a = b$ ahora, entonces a siempre será igual a b. En Python, una sentencia de asignación puede lograr que dos variables sean iguales pero sólo por un tiempo determinado:

```
a = 5
b = a    # a y b ahora son iguales
a = 3    # a y b no son iguales ahora
```

La tercera línea cambia el valor de `a` pero no cambia el valor de `b`, así que ya no serán iguales. En algunos lenguajes de programación se usa un signo diferente para la asignación como `<-` o `:=` para evitar la confusión.
Aunque la asignación múltiple puede ser útil se debe usar con precaución. Si los valores de las variables cambian frecuentemente se puede dificultar la lectura y la depuración del código.

7.2. La sentencia `while (mientras)`

Los computadores se usan a menudo para automatizar tareas repetitivas. Esto es algo que los computadores hacen bien y los seres humanos hacemos mal.
Hemos visto dos programas, `nLineas` y `conteo`, que usan la recursión para lograr repetir, lo que también se denomina **iteración**. Como la iteración es tan común, Python proporciona varios elementos para facilitarla. El primero que veremos es la sentencia `while`.
Aquí presentamos a la función `conteo` usando una sentencia `while`:

```
def conteo(n):
  while n > 0:
    print n
    n = n-1
  print "Despegue!"
```

Como eliminamos el llamado recursivo, esta función deja de ser recursiva.
La sentencia `while` se puede leer como en el lenguaje natural. Quiere decir, "Mientras `n` sea mayor que 0, continúe desplegando el valor de `n` y reduciendo el valor de `n` en 1. Cuando llegue a 0, despliegue la cadena `Despegue!`".
Más formalmente, el flujo de ejecución de una sentencia `while` luce así:

1. Evalúa la condición, resultando en `False` (falso) o `True` (cierto).

2. Si la condición es falsa (False), se sale de la sentencia `while` y continúa la ejecución con la siguiente sentencia (afuera del while).

3. Si la condición es cierta (True), ejecute cada una de las sentencias en el cuerpo y regrese al paso 1.

El cuerpo comprende todas las sentencias bajo la cabecera que tienen la misma indentación.

Este flujo se denomina **ciclo** porque el tercer paso da la vuelta hacia el primero. Note que si la condición es falsa la primera vez que se entra al while, las sentencias internas nunca se ejecutan.

El cuerpo del ciclo debería cambiar el valor de una o más variables, de forma que la condición se haga falsa en algún momento y el ciclo termine. De otra forma, el ciclo se repetirá para siempre, obteniendo un **ciclo infinito**. Una broma común entre los científicos de la computación es interpretar las instrucciones de los champús, "Aplique champú, aplique rinse, repita," como un ciclo infinito.

En el caso de `conteo`, podemos probar que el ciclo termina porque sabemos que el valor de `n` es finito, y podemos ver que va haciéndose más pequeño cada vez que el while itera (da la vuelta), así que eventualmente llegaremos a 0. En otros casos esto no es tan fácil de asegurar:

```
def secuencia(n):
  while n != 1:
    print n,
    if n%2 == 0:        # n es par
      n = n/2
    else:               # n es impar
      n = n*3+1
```

La condición para este ciclo es `n != 1`, así que se repetirá hasta que `n` sea `1`, lo que hará que la condición sea falsa.

En cada iteración del ciclo while el programa despliega el valor de `n` y luego chequea si es par o impar. Si es par, el valor de `n` se divide por 2. Si es impar el valor se reemplaza por `n*3+1`. Si el valor inicial (del argumento) es 3, la secuencia que resulta es 3, 10, 5, 16, 8, 4, 2, 1.

Como `n` aumenta algunas veces y otras disminuye, no hay una demostración obvia de que `n` llegará a ser 1, o de que el programa termina. Para algunos valores particulares de `n` podemos demostrar la terminación. Por ejemplo, si el valor inicial es una potencia de dos, entonces el valor de `n` será par en cada iteración del ciclo hasta llegar a 1. El ejemplo anterior termina con una secuencia así que empieza con 16.

Dejando los valores particulares de lado, la interesante pregunta que nos planteamos es si podemos demostrar que este programa termina para *todos* los valores de `n`. Hasta ahora, ¡nadie ha sido capaz de probarlo *o* refutarlo!.

> *Como ejericio, reescriba la función* `nLineas` *de la Sección 5.9, usando iteración en vez de recursión.*

7.3. Tablas

Una gama de aplicaciones, donde los ciclos se destacan, es la generación de información tabular. Antes de que los computadores existieran la gente tenía que calcular logaritmos, senos, cosenos y otras funciones matemáticas a mano. Para facilitar la tarea, los libros matemáticos incluían largas tablas con los valores de dichas funciones. La creación de las tablas era un proceso lento y aburridor, y tendían a quedar con muchos errores.

Cuando los computadores entraron en escena, una de las reacciones iniciales fue "Esto es maravilloso! Podemos usar los computadores para generar las tablas, de forma que no habrían errores". Eso resultó (casi) cierto, pero poco prospectivo. Poco después los computadores y las calculadoras se hicieron tan ubicuos que las tablas se hicieron obsoletas.

Bueno, casi. Para algunas operaciones los computadores usan tablas de valores para obtener una respuesta aproximada y luego hacer mas cálculos para mejorar la aproximación. En algunos casos, se han encontrado errores en las tablas subyacentes, el más famoso ha sido el de la tabla para realizar la división en punto flotante en los procesadores Pentium de la compañía Intel.

Aunque una tabla logarítmica no es tan útil como en el pasado, todavía sirve como un buen ejemplo de iteración. El siguiente programa despliega una secuencia de valores en la columna izquierda y sus logaritmos en la columna derecha:

```
x = 1.0
while x < 10.0:
  print x, '\t', math.log(x)
  x = x + 1.0
```

La cadena `'\t'` representa un carácter **tab** (tabulador).

A medida que los carácteres y las cadenas se despliegan en la pantalla un marcador invisible denominado **cursor** lleva pista de dónde va a ir el siguiente carácter. Después de una sentencia `print`, el cursor va al comienzo de la siguiente línea.

El carácter tabulador mueve el cursor hacia la derecha hasta que alcanza un punto de parada (cada cierto número de espacios, que pueden variar de sistema a sistema). Los tabuladores son útiles para alinear columnas de texto, como la salida del anterior programa:

```
1.0     0.0
2.0     0.69314718056
3.0     1.09861228867
4.0     1.38629436112
5.0     1.60943791243
6.0     1.79175946923
7.0     1.94591014906
```

```
8.0     2.07944154168
9.0     2.19722457734
```

Si estos valores parecen extraños, recuerde que la función `log` usa la base `e`. Ya que las potencias de dos son importantes en la ciencias de la computación, a menudo deseamos calcular logaritmos en base 2. Para este fin podemos usar la siguiente fórmula:

$$\log_2 x = \frac{log_e x}{log_e 2} \qquad (7.1)$$

Cambiando la salida del ciclo a:

```
print x, '\t',  math.log(x)/math.log(2.0)
```

resulta en:

```
1.0     0.0
2.0     1.0
3.0     1.58496250072
4.0     2.0
5.0     2.32192809489
6.0     2.58496250072
7.0     2.80735492206
8.0     3.0
9.0     3.16992500144
```

Podemos ver que 1, 2, 4, y 8 son potencias de dos porque sus logaritmos en base 2 son números enteros. Si deseamos calcular el logaritmo de más potencias de dos podemos modificar el programa así:

```
x = 1.0
while x < 100.0:
  print x, '\t', math.log(x)/math.log(2.0)
  x = x * 2.0
```

Ahora, en lugar de agregar algo a `x` en cada iteración del ciclo, produciendo una serie aritmética, multiplicamos a `x` por algo constante, produciendo una serie geométrica. El resultado es:

```
1.0     0.0
2.0     1.0
4.0     2.0
8.0     3.0
16.0    4.0
32.0    5.0
64.0    6.0
```

Gracias a los carácteres tabuladores entre las columnas, la posición de la segunda columna no depende del número de dígitos en la primera.
Puede que las tablas de logaritmos no sirvan en nuestros días, ¡pero para los científicos de la computación saber las potencias de dos sí es muy importante!.

> *Como ejercicio, modifique este programa de forma que despliegue las potencias de 2 hasta 65,536 (esto es 2^{16}). Imprima el resultado y familiaricese con las potencias de dos.*

El carácter diagonal invertido (backslash) `'\'` indica el comienzo de una **secuencia de escape**. Estas secuencias se utilizan para representar carácteres invisibles como tabuladores y nuevas líneas. La secuencia `'\n'` representa una nueva línea.
Una secuencia de escape puede empezar en cualquier posición de una cadena; en el ejemplo anterior, la secuencia de escape tabuladora es toda la cadena.
¿Cómo cree que se representa un diagonal invertido en una cadena?

> *Como ejercicio, escriba una sola cadena que*

```
produzca
        esta
                salida.
```

7.4. Tablas de dos dimensiones

Una tabla de dos dimensiones es una en la que los valores se leen en la intersección de una fila y una columna. Una tabla de multiplicación es un ejemplo familiar. Digamos que usted desea imprimir una tabla de multiplicación para los valores del 1 al 6.
Una buena forma de empezar es escribir un ciclo que imprima los múltiplos de 2, en una sola línea:

```
i = 1
while i <= 6:
  print 2*i, '   ',
  i = i + 1
print
```

La primera línea inicializa una variable llamada `i`, que actúa como un contador o **variable de ciclo**. A medida que se ejecuta el ciclo, el valor de `i` se incrementa de 1 a 6. Cuando `i` es 7, el ciclo termina. En cada iteración del ciclo despliega el valor `2*i`, seguido de tres espacios.

De nuevo, la coma en la sentencia `print` suprime la nueva línea. Después de que el ciclo termina la segunda sentencia `print` comienza una nueva línea.
La salida del programa es:

```
2       4       6       8       10      12
```

Hasta aquí vamos bien. El paso siguiente es **encapsular** y **generalizar**.

7.5. Encapsulamiento y generalización

Encapsular es el proceso de envolver un trozo de código en una función, permitiendo tomar ventaja de todas los beneficios que esto conlleva. Usted ha visto dos ejemplos de encapsulamiento: `imprimirParidad` en la Sección 5.5; y `esDivisible` en la Sección 6.4.
La generalización es tomar algo específico, tal como imprimir los múltiplos de 2, y convertirlo en algo más general, tal como imprimir los múltiplos de cualquier entero.
Esta función encapsula el ciclo anterior y lo generaliza para imprimir múltiplos de un parámetro `n`:

```
def imprimaMultiplos(n):
  i = 1
  while i <= 6:
    print n*i, '\t',
    i = i + 1
  print
```

Para encapsular el ciclo, todo lo que agregamos fue la primera línea que declara el nombre de la función y la lista de parámetros. Para generalizar, lo que hicimos fue reemplazar el valor 2 con el parámetro `n`.
Si llamamos a esta función con el argumento 2, obtenemos la misma salida anterior. Con el argumento 3, la salida es:

```
3       6       9       12      15      18
```

Con el argumento 4, la salida es:

```
4       8       12      16      20      24
```

Ahora, usted probablemente imagine cómo imprimir una tabla de multiplicación —llamando a `imprimaMultiplos` repetidamente con diferentes argumentos. De hecho, podemos usar otro ciclo:

```
i = 1
while i <= 6:
  imprimaMultiplos(i)
  i = i + 1
```

Observe lo similar que es este ciclo al ciclo interno de la función `imprimaMultiplos`. Todo lo que hicimos fue reemplazar la sentencia `print` con un llamado de función.
La salida de este programa es una tabla de multiplicación:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

7.6. Más encapsulamiento

Para demostrar el encapsulamiento otra vez, tomemos el código de la sección 7.5 y envolvámoslo en una función:

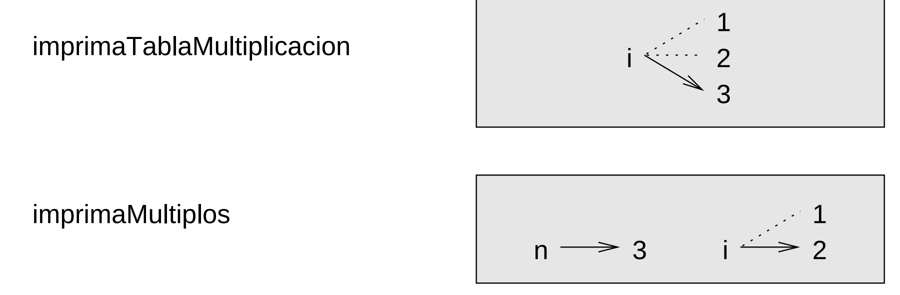
```
def imprimirTablaMultiplicacion():
  i = 1
  while i <= 6:
    imprimaMultiplos(i)
    i = i + 1
```

Este proceso es un **plan de desarrollo** común. Desarrollamos código escribiendo líneas afuera de cualquier función o digitándolas en el intérprete. Cuando funcionan, las ponemos dentro de una función.
Este plan de desarrollo es particularmente útil si usted no sabe, cuando está empezando a programar, cómo dividir el programa en funciones. Este enfoque permite diseñar a medida que se escribe código.

7.7. Variables locales

Puede estar preguntándose cómo usamos la misma variable, `i` en las dos funciones `imprimaMultiplos` y `imprimaTabla`. ¿ No ocurren problemas cuando una de ellas cambia el valor de la variable?
La respuesta es no, porque la `i` en `imprimaMultiplos` y la `i` en `imprimaTabla` *no* son la misma variable.

Las variables creadas dentro de una definición de función son locales; no se puede acceder a ellas fuera de su función "casa". Esto quiere decir que usted tiene la libertad de tener múltiples variables con el mismo nombre en tanto no pertenezcan a la misma función.
El diagrama de pila para este programa muestra que las dos variables llamadas `i` no son la misma. Se pueden referir a valores diferentes, y cambiar una de ellas no altera la otra.



El valor de `i` en `imprimaTabla` va de 1 a 6. En el diagrama va por el valor 3. En la próxima iteración del ciclo será 4. En cada iteración, `imprimaTabla` llama a `imprimaMultiplos` con el valor actual de `i` como argumento. Ese valor se le asigna al parámetro `n`.
Adentro de `imprimaMultiplos` el valor de `i` va de 1 a 6. En el diagrama, es 2. Cambiar esta variable no tiene efecto en el valor de `i` en `imprimaTabla`.
Es legal y muy común tener diferentes variables locales con el mismo nombre. Particularmente, nombres como `i` y `j` se usan frecuentemente como variables de ciclo. Si usted evita usarlas en una función porque ya las usó en otra, probablemente dificultará la lectura del programa.

7.8. Mas generalización

Como otro ejemplo de generalización, imagine que le piden un programa que imprima una tabla de multiplicación de cualquier tamaño; no sólo la tabla de seis-por-seis. Podría agregar un parámetro a `imprimaTabla`:

```
def imprimaTabla(tope):
  i = 1
  while i <= tope:
    imprimaMultiplos(i)
    i = i + 1
```

Reemplazamos el valor 6 con el parámetro `tope`. Si llamamos a `imprimaTabla` con el argumento 7, despliega:

```
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
7       14      21      28      35      42
```

Esto está bien, pero quizás deseamos que la tabla sea cuadrada—con el mismo número de filas y columnas. Para lograrlo, añadimos un parámetro a imprimaMultiplos que especifique cuántas columnas debe tener la tabla.
Sólo por confundir, vamos a nombrarlo `tope`, demostrando que diferentes funciones pueden tener parámetros con el mismo nombre (igual que las variables locales). Aquí está el programa completo:

```
def imprimaMultiplos(n, tope):
  i = 1
  while i <= tope:
    print n*i, '\t',
    i = i + 1
  print

def imprimaTabla(tope):
  i = 1
  while i <= tope:
    imprimaMultiplos(i, tope)
    i = i + 1
```

Note que cuando agregamos el nuevo parámetro tuvimos que cambiar la primera línea de la función (la cabecera), y también el lugar donde la función se llama en `imprimaTabla`.
Como se esperaba, este programa genera una tabla cuadrada de siete-por-siete:

```
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
```

Cuando se generaliza una función adecuadamente, a menudo se obtiene un programa con capacidades que no se habían planeado. Por ejemplo, podríamos apro-

vechar el hecho de que $ab = ba$, que causa que todas las entradas de la tabla aparezcan dos veces para ahorrar tinta imprimiendo solamente la mitad de la tabla. Para lograrlo sólo se necesita cambiar una línea de `imprimaTabla`. Cambiamos:

```
imprimaMultiplos(i, tope)
```

por

```
imprimaMultiplos(i, i)
```

y se obtiene:

```
1
2       4
3       6       9
4       8       12      16
5       10      15      20      25
6       12      18      24      30      36
7       14      21      28      35      42      49
```

Como ejercicio, siga la ejecución de esta versión de `imprimaTabla` *y descifre cómo trabaja.*

7.9. Funciones

Ya hemos mencionado los "beneficios de las funciones." Usted se estará preguntado cuales son estos beneficios. Aquí hay algunos:

- Nombrar una secuencia de sentencias aumenta la legibilidad de los programas y los hace más fáciles de depurar.
- Dividir un programa grande en funciones permite separar partes de éste, depurarlas aisladamente, y luego componerlas en un todo coherente.
- Las funciones facilitan la recursión y la iteración.
- Las funciones bien diseñadas resultan ser útiles para muchos programas. Una vez que usted escribe y depura una, se puede reutilizar.

7.10. Glosario

Asignación múltiple: realizar más de una asignación a una misma variable durante la ejecución de un programa.

Iteración: la ejecución repetida de un grupo de sentencias, ya sea en una función recursiva o en un ciclo.

Ciclo: una sentencia o grupo de sentencias que se ejecuta repetidamente hasta que una condición de terminación se cumple.

Ciclo infinito: ciclo en el que la condición de terminación nunca se cumple.

Cuerpo: las sentencias adentro de un ciclo.

Variable de ciclo: variable que se usa como parte de la condición de terminación de un ciclo.

Tab: (tabulador) carácter especial que causa el movimiento del cursor al siguiente punto de parada en la línea actual.

Nueva línea: carácter que causa que el cursor se mueva al principio de la siguiente línea.

Cursor: marcador invisible que lleva pista de dónde se va a imprimir el siguiente carácter.

Secuencia de escape: carácter de escape (\) seguido por uno o más carácteres imprimibles que se usan para denotar un carácter no imprimible.

Encapsular: dividir un programa grande y complejo en componentes (como funciones) y aislarlos entre sí (usando variables locales, por ejemplo).

Generalizar: reemplazar algo innecesariamente específico (como un valor constante) con algo general más apropiado (como una variable o parámetro). La generalización aumenta la versatilidad del código, lo hace más reutilizable y en algunos casos facilita su escritura.

Plan de desarrollo: proceso para desarrollar un programa. En este capítulo demostramos un estilo de desarrollo basado en escribir código que realiza cómputos simples y específicos, para luego encapsularlo y generalizarlo.

Capítulo 8

Cadenas

8.1. Un tipo de dato compuesto

Hasta aquí hemos visto tres tipos de datos: `int`, `float` y `string`. Las cadenas son cualitativamente diferentes de los otros dos tipos porque están compuestas de piezas más pequeñas—los carácteres.

Los tipos que comprenden piezas más pequeñas se denominan **tipos de datos compuestos**. Dependiendo de lo que hagamos podemos tratar un tipo compuesto como unidad o podemos acceder a sus partes. Esta ambigüedad es provechosa.

El operador corchete selecciona un carácter de una cadena.

```
>>> fruta = "banano"
>>> letra = fruta[1]
>>> print letra
```

La expresión `fruta[1]` selecciona el carácter número 1 de `fruta`. La variable `letra` se refiere al resultado. Cuando desplegamos `letra`, obtenemos una pequeña sorpresa:

```
a
```

La primera letra de `"banano"` no es `a`. ¡A menos que usted sea un científico de la computación!.Por razones perversas, los científicos de la computación empiezan a contar desde cero. La letra número 0 de `"banano"` es `b`. La letra 1 es `a`, y la letra 2 es `n`.

Si usted desea la primera letra de una cadena se pone 0, o cualquier expresión con el valor 0, dentro de los corchetes:

```
>>> letra = fruta[0]
>>> print letra
b
```

La expresión en corchetes se denomina **índice**. Un índice especifica un miembro de un conjunto ordenado, en este caso el conjunto de carácteres de la cadena. El índice *indica* cual elemento desea usted, por eso se llama así. Puede ser cualquier expresión entera.

8.2. Longitud

La función `len` retorna el número de carácteres en una cadena:

```
>>> fruta = "banano"
>>> len(fruta)
6
```

Para acceder a la última letra de una cadena usted podría caer en algo como esto:

```
longitud = len(fruta)
ultima = fruta[longitud]       # ERROR!
```

Y no funcionará. Causa un error en tiempo de ejecución, `IndexError: string index out of range`. La razón yace en que no hay una letra 6 en `"banana"`. Como empezamos a contar desde cero, las seis letras se numeran de 0 a 5. En general, para obtener la última letra, tenemos que restar 1 a la `longitud`:

```
longitud = len(fruta)
ultima = fruta[longitud-1]
```

Alternativamente, podemos usar índices negativos, que cuentan hacia atrás desde el fin de la cadena. La expresión `fruta[-1]` retorna la última letra `fruit[-2]` retorna la penúltima, y así sucesivamente.

8.3. Recorridos en cadenas y el ciclo `for`

Muchos cálculos implican procesar una cadena carácter por carácter. A menudo empiezan al inicio, seleccionan cada carácter en cada paso, le hacen algo y continúan hasta el final. Este patrón de procesamiento se denomina **recorrido**. Hay una forma de realizarlo con la sentencia `while`:

```
indice = 0
while indice < len(fruta):
  letra = fruta[indice]
  print letra
  indice = indice + 1
```

Este ciclo recorre la cadena y despliega cada letra en una línea independiente. La condición del ciclo es `indice <len(fruta)`, así que cuando `indice` se hace igual a la longitud de la cadena, la condición es falsa, y el cuerpo del ciclo no se ejecuta. El último carácter accedido es el que tiene el índice `len(fruta)-1`, es decir, el último.

> *Como ejercicio, escriba una función que tome una cadena como argumento y despliegue las letras al revés, una por cada línea.*

Usar un índice para recorrer un conjunto de valores es tan común que Python tiene una sintaxis alternativa más simple—el ciclo `for` :

```
for carácter in fruta:
  print carácter
```

Cada vez que el ciclo itera, el próximo carácter en la cadena se asigna a la variable `carácter`. El ciclo continúa hasta que no quedan más carácteres.
El siguiente ejemplo muestra cómo usar la concatenación y un ciclo `for` para generar una serie en orden lexicográfico. Lexicográfico se refiere a una lista en la que los elementos aparecen en orden alfabético. Por ejemplo, en el libro *Make Way for Ducklings* de Robert McCloskey, los nombres de los patos son Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. Este ciclo los despliega en orden:

```
prefijos = "JKLMNOPQ"
sufijo = "ack"

for letra in prefijos:
  print letra + sufijo
```

La salida de este programa es:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Por supuesto que hay un error, ya que "Ouack" y "Quack" no están bien deletreados.

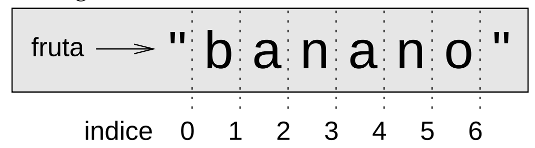
> *Como ejercicio, modifique el programa para corregir este error.*

8.4. Segmentos de cadenas

Una porción de una cadena de caracteres se denomina **segmento**. Seleccionar un segmento es similar a seleccionar un carácter:

```
>>> s = "Pedro, Pablo, y Maria"
>>> print s[0:5]
Pedro
>>> print s[7:12]
Pablo
>>> print s[16:21]
Maria
```

El operador `[n:m]` retorna la parte de la cadena que va desde el carácter n hasta el m, incluyendo el primero y excluyendo el último. Este comportamiento es contraintuitivo, tiene más sentido si se imagina que los índices van *antes* de los carácteres, como en el siguiente diagrama:



Si usted omite el primer índice (despues de los puntos seguidos), el segmento comienza en el inicio de la cadena. Si se omite el segundo índice, el segmento va hasta el final. Entonces:

```
>>> fruta = "banana"
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ano'
```

¿Que cree que significa `s[:]`?

8.5. Comparación de cadenas

El operador de comparación funciona con cadenas. Para ver si dos cadenas son iguales:

```
if palabra == "banana":
  print  "No hay bananas!"
```

Las otras operaciones de comparación son útiles para poner las palabras en orden alfabético:

```
if palabra < "banana":
  print "Su palabra," + palabra + ", va antes que banana."
elif palabra > "banana":
  print "Su palabra," + palabra + ", va después de banana."
else:
  print "No hay bananas!"
```

Sin embargo, usted debe ser consciente de que Python no maneja las letras minúsculas y mayúsculas de la misma forma en que lo hace la gente. Todas las letras mayúsculas vienen antes de las minúsculas. Si palabra vale "Zebra"la salida sería:

```
Su palabra, Zebra, va antes que banana.
```

Este problema se resuelve usualmente convirtiendo las cadenas a un formato común, todas en minúsculas por ejemplo, antes de hacer la comparación. Un problema más difícil es lograr que el programa reconozca que una zebra no es una fruta.

8.6. Las cadenas son inmutables

Uno puede caer en la trampa de usar el operador `[]` al lado izquierdo de una asignación con la intención de modificar un carácter en una cadena. Por ejemplo:

```
saludo = "Hola mundo"
saludo[0] = 'J'            # ERROR!
print saludo
```

En lugar de desplegar `Jola mundo!`, se produce un error en tiempo de ejecución `TypeError: object doesn't support item assignment`.
Las cadenas son **inmutables**, lo que quiere decir que no se puede cambiar una cadena existente. Lo máximo que se puede hacer es crear otra cadena que cambia un poco a la original:

```
saludo = "Hola mundo!"
nuevoSaludo = 'J' + saludo[1:]
print nuevoSaludo
```

La solución consiste en concatenar la primera nueva letra con un segmento de `saludo`. Esto no tiene efecto sobre la primera cadena, usted puede chequearlo.

8.7. Una función buscar

¿Qué hace la siguiente función?

```
def buscar(cad, c):
  indice = 0
  while indice < len(cad):
    if cad[indice] == c:
      return indice
    indice = indice + 1
  return -1
```

De cierta manera `buscar` es el opuesto del operador `[]`. En vez de tomar un índice y extraer el carácter correspondiente, toma un carácter y encuentra el índice donde éste se encuentra. Si no se encuentra el carácter en la cadena, la función retorna `-1`.
Este es el primer ejemplo de una sentencia `return` dentro de un ciclo. Si se cumple que `cadena[indice] == c`, la función retorna inmediatamente, rompiendo el ciclo prematuramente.
Si el carácter no está en la cadena, el programa completa todo el ciclo y retorna `-1`.
Este patrón computacional se denomina recorrido "eureka", ya que tan pronto encontremos lo que buscamos, gritamos "Eureka!" y dejamos de buscar.

> *Como ejercicio, modifique la función* `buscar` *de forma que reciba un tercer parámetro, el índice en la cadena donde debe empezar a buscar.*

8.8. Iterando y contando

El siguiente programa cuenta el número de veces que la letra `a` aparece en una cadena:

```
fruta = "banano"
cont = 0
for car in fruta:
  if car == 'a':
    cont = cont + 1
print cont
```

Este programa demuestra otro patrón computacional denominado **contador**. La variable `cont` se inicializa a 0 y se incrementa cada vez que se encuentre una `a`. (**incrementar** es añadir uno; es el opuesto de **decrementar**, y no tienen nada que ver con "excremento," que es un sustantivo.) Cuando el ciclo finaliza, `cont` contiene el resultado—el número total de `a`'s.

Como ejercicio, encapsule este código en una función llamada `contarLetras`*, y generalícela de forma que reciba la cadena y la letra como parámetros.*

Otro ejercicio, reescriba esta función de forma que en lugar de recorrer la cadena, llame a la función `buscar` *anterior que recibe tres parámetros.*

8.9. El módulo string

El módulo `string` contiene funciones útiles para manipular cadenas. Como de costumbre, tenemos que importarlo antes de usarlo:

```
>>> import string
```

El módulo `string` incluye una función denominada `find` que hace lo mismo que buscar. Para llamarla tenemos que especificar el nombre del módulo y de la función, usando la notación punto.

```
>>> fruta = "banano"
>>> ind = string.find(fruta, "a")
>>> print ind
1
```

Uno de los beneficios de los módulos es que ayudan a evitar colisiones entre los nombres de las funciones primitivas y los nombres de las funciones creadas por el programador. Si hubiéramos nombrado a nuestra función buscar con la palabra inglesa `find`, podríamos usar la notación punto para especificar que queremos llamar a la función find del módulo string, y no a la nuestra.
De hecho `string.find` es más general que buscar, también puede buscar subcadenas, no solo carácteres:

```
>>> string.find("banano", "na")
2
```

También tiene un argumento adicional que especifica el índice desde el que debe empezar la búsqueda:

```
>>> string.find("banana", "na", 3)
4
```

Igualmente, puede tomar dos argumentos adicionales que especifican un rango de índices:

```
>>> string.find("bob", "b", 1, 2)
-1
```

Aquí la búsqueda falló porque la letra *b* no está en en el rango de índices de `1` a `2` (recuerde que no se incluye el último índice, el `2`).

8.10. Clasificación de carácteres

Con frecuencia es útil examinar un carácter y decidir si está en mayúsculas o en minúsculas, o si es un dígito. El módulo `string` proporciona varias constantes que sirven para lograr estos objetivos.
La cadena `string.lowercase` contiene todas las letras que el sistema considera como minúsculas. Igualmente, `string.uppercase` contiene todas las letras mayúsculas. Intente lo siguiente y vea por sí mismo:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

Podemos usar estas constantes y la función `find` para clasificar los carácteres. Por ejemplo, si `find(lowercase, c)` retorna un valor distinto de `-1`, entonces `c` debe ser una letra minúscula:

```
def esMinuscula(c):
  return find(string.lowercase, c) != -1
```

Otra alternativa la da el operador `in` que determina si un carácter aparece en una cadena:

```
def esMinuscula(c):
  return c in string.lowercase
```

Y otra alternativa más, con el operador de comparación:

```
def esMinuscula(c):
  return 'a' <= c <= 'z'
```

Si `c` está entre *a* y *z*, debe ser una letra minúscula.

> *Como ejercicio, discuta qué versión de* `esMinuscula` *cree que es la más rápida. ¿Puede pensar en otra razón distinta de la velocidad para preferir alguna de ellas sobre las otras?*

Otra constante definida en el módulo `string` puede sorprenderlo cuando la imprima:

```
>>> print string.whitespace
```

Un carácter de los que pertenecen a **Whitespace** mueve el cursor sin imprimir nada. Crean un espacio en blanco que se puede evidenciar entre carácteres. La constante `string.whitespace` contiene todos los carácteres que representan espacios en blanco: espacio, tab (`\t`), y nueva línea (`\n`).
Hay otras funciones útiles en el módulo string, pero este libro no es un manual de referencia. Para esto usted puede consultar la referencia de las bibliotecas de Python (*Python Library Reference*). Además, hay un gran cúmulo de documentación en el sitio web de Python `www.python.org`.

8.11. Glosario

Tipo de dato compuesto: un tipo de dato en el que los valores están compuestos por componentes o elementos, que, a su vez, son valores.

Recorrido: iteración sobre todos los elementos de un conjunto ejecutando una operación similar en cada uno.

Índice: variable o valor que se usa para seleccionar un miembro de un conjunto ordenado, tal como los carácteres de una cadena. También se puede usar el término `posición` como sinónimo de índice.

Segmento: parte de una cadena, especificada por un rango de índices.

Mutable: un tipo de dato compuesto a cuyos elementos pueden asignarseles nuevos valores.

Contador: una variable que se usa para contar algo, usualmente se inicializa en cero y se incrementa posteriormente dentro de un ciclo.

Incrementar: agregar uno al valor de una variable

Decrementar: restar uno al valor de una variable

Espacio en blanco: cualquiera de los carácteres que mueven el cursor sin imprimir nada visible. La constante `string.whitespace` contiene todos los carácteres que representan espacios en blanco.

Capítulo 9

Listas

Una **lista** es un conjunto ordenado de valores que se identifican por medio de un índice. Los valores que componen una lista se denominan **elementos**. Las listas son similares a las cadenas, que son conjuntos ordenados de carácteres, pero son mas generales, ya que pueden tener elementos de cualquier tipo de dato. Las listas y las cadenas—y otras conjuntos ordenados que veremos— se denominan **secuencias**.

9.1. Creación de listas

Hay varias formas de crear una nueva lista; la más simple es encerrar los elementos entre corchetes ([y]):

```
[10, 20, 30, 40]
["correo", "lapiz", "carro"]
```

El primer ejemplo es una lista de cuatro enteros, la segunda, una lista de tres cadenas. Los elementos de una lista no tienen que tener el mismo tipo. La siguiente lista contiene una cadena, un flotante, un entero y (mirabile dictu) otra lista:

```
["hola", 2.0, 5, [10, 20]]
```

Cuando una lista está contenida por otra se dice que está **anidada**.
Las listas que contienen enteros consecutivos son muy comunes, así que Python proporciona una forma de crearlas:

```
>>> range(1,5)
[1, 2, 3, 4]
```

La función range toma dos argumentos y retorna una lista que contiene todos los enteros desde el primero hasta el segundo, ¡incluyendo el primero y no el último!
Hay otras formas de usar a range. Con un solo argumento crea una lista que empieza en 0:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si hay un tercer argumento, este especifica el espacio entre los valores sucesivos, que se denomina el **tamaño del paso**. Este ejemplo cuenta de 1 a 10 con un paso de tamaño 2:

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Finalmente, existe una lista especial que no contiene elementos. Se denomina lista vacía, y se denota con [].
Con todas estas formas de crear listas sería decepcionante si no pudiéramos asignar listas a variables o pasarlas como parámetros a funciones. De hecho, podemos hacerlo:

```
>>> vocabulario = ["mejorar", "castigar", "derrocar"]
>>> numeros = [17, 123]
>>> vacia = []
>>> print vocabulario, numeros, vacia
["mejorar", "castigar", "derrocar"] [17, 123] []
```

9.2. Accediendo a los elementos

La sintaxis para acceder a los elementos de una lista es la misma que usamos en las cadenas—el operador corchete ([]). La expresión dentro de los corchetes especifica el índice. Recuerde que los índices o posiciones empiezan desde 0:

```
print numeros[0]
numeros[1] = 5
```

El operador corchete para listas puede aparecer en cualquier lugar de una expresión. Cuanto aparece al lado izquierdo de una asignación cambia uno de los elementos de la lista de forma que el elemento 1 de numeros, que tenía el valor 123, ahora es 5.
Cualquier expresión entera puede usarse como índice:

```
>>> numeros[3-2]
5
>>> numeros[1.0]
TypeError: sequence index must be integer
```

Si usted intenta leer o escribir un elemento que no existe, obtiene un error en tiempo de ejecución:

```
>>> numeros[2] = 5
IndexError: list assignment index out of range
```

Si el índice tiene un valor negativo, cuenta hacia atrás desde el final de la lista:

```
>>> numeros[-1]
5
>>> numeros[-2]
17
>>> numeros[-3]
IndexError: list index out of range
```

`numeros[-1]` es el último elemento de la lista, `numeros[-2]` es el penúltimo, y `numeros[-3]` no existe.
Usualmente se usan variables de ciclo como índices de listas:

```
combatientes = ["guerra", "hambruna", "peste", "muerte"]

i = 0
while i < 4:
  print combatientes[i]
  i = i + 1
```

Este ciclo `while` cuenta de 0 a 4. Cuando la variable de ciclo `i` es 4, la condición falla y el ciclo termina. El cuerpo del ciclo se ejecuta solamente cuando `i` es 0, 1, 2, y 3.
En cada iteración del ciclo, la variable `i` se usa como un índice a la lista, imprimiendo el `i`-ésimo elemento. Este patrón se denomina **recorrido de una lista**.

9.3. Longitud de una lista

La función `len` retorna la longitud de una lista. Es una buena idea usar este valor como límite superior de un ciclo en vez de una constante. De ésta forma, si la lista cambia, usted no tendrá que cambiar todos los ciclos del programa, ellos funcionarán correctamente para listas de cualquier tamaño:

```
combatientes = ["guerra", "hambruna", "peste", "muerte"]

i = 0
while i < len(combatientes):
  print combatientes[i]
  i = i + 1
```

La última vez que el ciclo se ejecuta `i` es `len(combatientes) - 1`, que es la posición del último elemento. Cuando `i` es igual a `len(combatientes)`, la condición falla y el cuerpo no se ejecuta, lo que está muy bien , ya que `len(combatientes)` no es un índice válido.
Aunque una lista puede contener a otra, la lista anidada se sigue viendo como un elemento único. La longitud de esta lista es cuatro:

```
['basura!', 1, ['Brie', 'Roquefort', 'Pol le Veq'],
 [1, 2, 3]]
```

Como ejercicio, escriba un ciclo que recorra la lista anterior imprimiendo la longitud de cada elemento. ¿Qué pasa si usted le pasa un entero a `len`?

9.4. Pertenencia

`in` es un operador booleano que chequea la pertenencia de un valor a una secuencia. Lo usamos en la Sección 8.10 con cadenas, pero también funciona con listas y otras secuencias:

```
>>> combatientes = ["guerra", "hambruna", "peste", "muerte"]
>>> 'peste' in combatientes
True
>>> 'corrupcion' in combatientes
False
```

Ya que "peste" es un miembro de la lista `combatientes`, el operador `in` retorna cierto. Como "corrupcion" no está en la lista, `in` retorna falso.
Podemos usar el operador lógico `not` en combinación con el `in` para chequear si un elemento no es miembro de una lista:

```
>>> 'corrupcion' not in combatientes
True
```

9.5. Listas y ciclos for

El ciclo `for` que vimos en la Sección 8.3 también funciona con listas. La sintaxis generalizada de un ciclo `for` es:

```
for VARIABLE in LISTA:
  CUERPO
```

Esto es equivalente a:

```
i = 0
while i < len(LISTA):
  VARIABLE = LISTA[i]
  CUERPO
  i = i + 1
```

El ciclo `for` es más conciso porque podemos eliminar la variable de ciclo `i`. Aquí está el ciclo de la sección anterior escrito con un `for` en vez de un while:

```
for combatiente in combatientes:
  print combatiente
```

Casi se lee como en español: "Para (cada) combatiente en (la lista de) combatientes, imprima (el nombre del) combatiente".
Cualquier expresión que cree una lista puede usarse en un ciclo `for`:

```
for numero in range(20):
  if numero % 2 == 0:
    print  numero

for fruta in ["banano", "manzana", "pera"]:
  print "Me gustaria comer " + fruta + "s!"
```

El primer ejemplo imprime todos los números pares entre uno y diecinueve. El segundo expresa entusiasmo sobre varias frutas.

9.6. Operaciones sobre listas

El operador + concatena listas:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarmente, el operador * repite una lista un número de veces determinado:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

El primer ejemplo repite `[0]` cuatro veces. El segundo repite `[1, 2, 3]` tres veces.

9.7. Segmentos de listas

Las operaciones para sacar segmentos de cadenas que vimos en la Sección 8.4 también funcionan con listas:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3]
['b', 'c']
>>> lista[:4]
['a', 'b', 'c', 'd']
>>> lista[3:]
['d', 'e', 'f']
>>> lista[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

9.8. Las listas son mutables

Las listas son mutables y no tienen la restricción de las cadenas, esto quiere decir que podemos cambiar los elementos internos usando el operador corchete al lado izquierdo de una asignación.

```
>>> fruta = ["banano", "manzana", "pera"]
>>> fruta[0] = "mandarina"
>>> fruta[-1] = "naranja"
>>> print fruta
['mandarina', 'manzana', 'naranja']
```

Con el operador segmento podemos actualizar varios elementos a la vez:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = ['x', 'y']
>>> print lista
['a', 'x', 'y', 'd', 'e', 'f']
```

También podemos eliminar varios elementos asignándoles la lista vacía:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> lista[1:3] = []
>>> print lista
['a', 'd', 'e', 'f']
```

Igualmente, podemos agregar elementos a una lista apretándolos dentro de un segmento vacío en la posición que deseamos:

```
>>> lista = ['a', 'd', 'f']
>>> lista[1:1] = ['b', 'c']
>>> print lista
['a', 'b', 'c', 'd', 'f']
>>> lista[4:4] = ['e']
>>> print lista
['a', 'b', 'c', 'd', 'e', 'f']
```

9.9. Otras operaciones sobre listas

Usar segmentos para insertar y borrar elemenos de una lista es extraño y propenso a errores. Hay mecanismos alternativos más legibles como `del` que elimina un elemento de una lista.

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

Como es de esperar, `del` recibe índices negativos, y causa errores en tiempo de ejecución si el índice está fuera de rango.
También se puede usar un segmento como argumento a `del`:

```
>>> lista = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del lista[1:5]
>>> print lista
['a', 'f']
```

Como de costumbre, los segmentos seleccionan todos los elementos hasta el segundo índice, sin incluirlo.
La función `append` agrega un elemento (o una lista) al final de una lista existente:

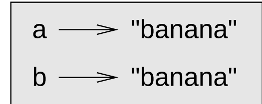
```
>>> a = ['uno', 'dos']
>>> a.append('tres')
>>> print a
```

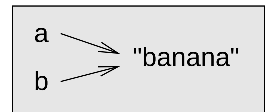
9.10. Objetos y valores

Si ejecutamos estas asignaciones

```
a = "banana"
b = "banana"
```

sabemos que `a` y `b` se referirán a una cadena con las letras `''banana''`. Pero no podemos afirmar que sea la *misma* cadena.
Hay dos situaciones posibles:





En un caso, `a` y `b` se refieren a cosas distintas que tienen el mismo valor. En el segundo caso, se refieren a la misma cosa. Estas "cosas" tienen nombres—se denominan **objetos**. Un objeto es algo a lo que se puede referir una variable.
Cada objeto tiene un **identificador** único, que podemos obtener con la función `id`. Imprimiendo el identificador de `a` y `b`, podemos saber si se refieren al mismo objeto.

```
>>> id(a)
135044008
>>> id(b)
135044008
```

De hecho, obtenemos el mismo identificador dos veces, lo que nos dice que Python sólo creó una cadena, y que `a` y `b` se refieren a ella.
Las listas, por otro lado, se comportan de manera diferente. Cuando creamos dos listas obtenemos dos objetos:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

Así que el diagrama de estados luce así:

```
a ──> [ 1, 2, 3 ]
b ──> [ 1, 2, 3 ]
```

a y b tienen el mismo valor pero no se refieren al mismo objeto.

9.11. Alias

Como las variables se pueden referir a objetos, si asignamos una variable a otra, las dos se referirán al mismo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
```

En este caso el diagrama de estados luce así:

```
a ─┐
   ├─> [ 1, 2, 3 ]
b ─┘
```

Como la misma lista tiene dos nombres distintos, a y b, podemos decir que b es un **alias** de a. Los cambios que se hagan a través de un alias afectan al otro:

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Aunque este comportamiento puede ser útil, algunas veces puede ser indeseable. En general, es más seguro evitar los alias cuando se está trabajando con objetos mutables. Para objetos inmutables no hay problema. Esta es la razón por la que Python tiene la libertad de crear alias a cadenas cuando ve la oportunidad de economizar memoria. Pero tenga en cuenta que esto puede variar en las diferentes versiones de Python; por lo tanto no es recomendable realizar programas que dependan de este comportamiento.

9.12. Clonando listas

Si queremos modificar una lista y conservar una copia de la original, necesitamos realizar una copia de la lista, no sólo de la referencia. Este proceso se denomina **clonación**, para evitar la ambigüedad de la palabra "copiar".
La forma más sencilla de clonar una lista es usar el operador segmento:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Al tomar cualquier segmento de `a` creamos una nueva lista. En este caso el segmento comprende toda la lista.
Ahora podemos realizar cambios a `b` sin preocuparnos por `a`:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

> *Como ejercicio, dibuje un diagrama de estados para* `a` *y* `b` *antes y después de este cambio.*

9.13. Listas como parámetros

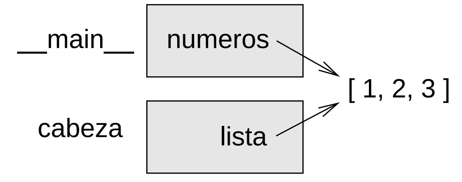
Pasar una lista como argumento es pasar una referencia, no una copia de ella. Por ejemplo, la función `cabeza` toma una lista como parámetro y retorna el primer elemento:

```
def cabeza(lista):
  return lista[0]
```

Se puede usar así:

```
>>> numeros = [1, 2, 3]
>>> cabeza(numeros)
1
```

El parámetro `lista` y la variable `numeros` son alias para el mismo objeto. El diagrama de estados luce así:



Como el objeto lista está compartido por dos marcos, lo dibujamos en el medio.
Si una función modifica un parámetro de tipo lista, el que hizo el llamado ve los cambios. Por ejemplo, `borrarCabeza` borra el primer elemento de una lista:

```
def borrarCabeza(lista):
  del lista[0]
```

Y se puede usar así::

```
>>> numeros = [1, 2, 3]
>>> borrarCabeza(numeros)
>>> print numeros
[2, 3]
```

Si una función retorna una lista, retorna una referencia a ella. Por ejemplo, la función `cola` retorna una lista que contiene todos los elementos, excepto el primero:

```
def cola(lista):
  return lista[1:]
```

`cola` se puede usar así:

```
>>> numeros = [1, 2, 3]
>>> resto = cola(numeros)
>>> print resto
[2, 3]
```

Como el valor de retorno se creó con el operador segmento, es una nueva lista. La creación de `resto`, y los cambios subsecuentes sobre esta variable no tienen efecto sobre `numeros`.

9.14. Listas anidadas

Una lista anidada aparece como elemento dentro de otra lista. En la siguiente lista, el tercer elemento es una lista anidada:

```
>>> lista = ["hola", 2.0, 5, [10, 20]]
```

Si imprimimos `lista[3]`, vemos `[10, 20]`. Para tomar un elemento de la lista anidada podemos realizar dos pasos:

```
>>> elt = lista[3]
>>> elt[0]
10
```

O, los podemos combinar:

```
>>> lista[3][1]
20
```

Las aplicaciones del operador corchete se evalúan de izquierda a derecha, así que ésta expresión obtiene el elemento 3 de `lista` y extrae de allí el elemento 1.

9.15. Matrices

Las listas anidadas se usan a menudo para representar matrices. Por ejemplo, la matriz:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

se puede representar así:

```
>>> matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`matriz` es una lista con tres elementos, cada uno es una fila. Podemos seleccionar una fila de la manera usual:

```
>>> matriz[1]
[4, 5, 6]
```

O podemos extraer un elemento individual de la matriz usando dos índices:

```
>>> matriz[1][1]
5
```

El primero escoge la fila, y el segundo selecciona la columna. Aunque esta forma de representar matrices es común, no es la única posibilidad. Una pequeña variación consiste en usar una lista de columnas en lugar de una lista de filas. Más adelante veremos una alternativa más radical, usando un diccionario.

9.16. Cadenas y listas

Dos de las funciones más usadas del módulo `string` implican listas de cadenas. `split` separa una cadena en una lista de palabras. Por defecto, cualquier número de espacios en blanco sirven como criterio de separación:

```
>>> import string
>>> cancion = "La vida es un ratico..."
>>> string.split(cancion)
['La', 'vida', 'es', 'un', 'ratico...']
```

Un argumento opcional denominado **delimitador** se puede usar para especificar que carácteres usar como criterio de separación. El siguiente ejemplo usa la cadena `an` como delimitador:

```
>>> string.split( "La rana que canta", "an")
['La r', 'a que c', 'ta']
```

Note que el delimitador no aparece en la lista resultante.
La función `join` es la inversa de `split`. Toma una lista de cadenas y las concatena con un espacio entre cada par:

```
>>> m = ['La', 'vida', 'es', 'un', 'ratico']
>>> string.join(m)
'La vida es un ratico'
```

Como `split`, `join` puede recibir un argumento adicional separador que se inserta entre los elementos:

```
>>> string.join(m, '_')
'La_vida_es_un_ratico'
```

> *Como ejercicio, describa la relación entre las expresiones* `cadena` *y* `string.join(string.split(cadena))`. *¿Son iguales para todas las cadenas? ¿Cuando serían diferentes?*

9.17. Glosario

Lista: colección de objetos que recibe un nombre. Cada objeto se identifica con un índice o número entero positivo.

Índice: valor o variable entero que indica la posición de un elemento en una lista.

Elemento: uno de los valores dentro de una lista (u otra secuencia). El operador corchete selecciona elementos de una lista.

Secuencia: los tipos de datos que contienen un conjunto ordenado de elementos, identificados por índices.

Lista anidada: lista que es elemento de otra lista.

Recorrido de una lista: es el acceso secuencial de cada elemento de una lista.

Objeto: una cosa a la que una variable se puede referir.

Alias: cuando varias variables tienen referencias hacia el mismo objeto.

Clonar: crear un objeto con el mismo valor que un objeto preexistente. Copiar una referencia a un objeto crea un alias, pero no clona el objeto.

Delimitador: carácter o cadena que se usa para indicar el lugar donde una cadena debe ser separada.

Capítulo 10

Tuplas

10.1. Mutabilidad y tuplas

Hasta aquí, usted ha visto dos tipos de datos compuestos: cadenas, que están compuestas de carácteres; y listas, compuestas de elementos de cualquier tipo. Una de las diferencias que notamos es que los elementos de una lista pueden modificarse, pero los carácteres en una cadena no. En otras palabras, las cadenas son **inmutables** y las listas son **mutables**.
Hay otro tipo de dato en Python denominado **tupla**, que se parece a una lista, con la excepción de que es inmutable. Sintácticamente, una tupla es una lista de valores separados por comas:

```
>>> tupla = 'a', 'b', 'c', 'd', 'e'
```

Aunque no es necesario, se pueden encerrar entre paréntesis:

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
```

Para crear una tupla con un único elemento, tenemos que incluir la coma final:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Sin la coma, Python creería que `('a')` es una cadena en paréntesis:

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

Las operaciones sobre tuplas son las mismas que vimos con las listas. El operador corchete selecciona un elemento de la tupla.

```
>>> tupla = ('a', 'b', 'c', 'd', 'e')
>>> tupla[0]
'a'
```

Y el operador segmento selecciona un rango de elementos:

```
>>> tupla[1:3]
('b', 'c')
```

Pero si intentamos modificar un elemento de la tupla obtenemos un error:

```
>>> tupla[0] = 'A'
TypeError: object doesn't support item assignment
```

Aunque no podemos modificar los elementos, sí podemos modificar toda la tupla:

```
>>> tupla = ('A',) + tupla[1:]
>>> tupla
('A', 'b', 'c', 'd', 'e')
>>> tupla = (1,2,3)
>>> tupla
```

10.2. Asignación de tuplas

De vez en cuando necesitamos intercambiar los valores de dos variables. Con el operador de asignación normal tenemos que usar una variable temporal. Por ejemplo, para intercambiar `a` y `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Si tenemos que intercambiar variables muchas veces, el código tiende a ser engorroso. Python proporciona una forma de **asignación de tuplas** que resuelve este problema:

```
>>> a, b = b, a
```

El lado izquierdo es una tupla de variables; el derecho es una tupla de valores. Cada valor se asigna a su respectiva variable en el orden en que se presenta. Las expresiones en el lado derecho se evalúan antes de cualquier asignación. Esto hace a la asignación de tuplas una herramienta bastante versátil.
Naturalmente, el número de variables a la izquierda y el número de valores a la derecha deben coincidir.

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```

10.3. Tuplas como valores de retorno

Las funciones pueden tener tuplas como valores de retorno. Por ejemplo, podríamos escribir una función que intercambie sus dos parámetros:

```
def intercambiar(x, y):
  return y, x
```

Así podemos asignar el valor de retorno a una tupla con dos variables:

```
a, b = intercambiar(a, b)
```

En este caso, escribir una función `intercambio` no es muy provechoso. De hecho, hay un peligro al tratar de encapsular `intercambio`, que consiste en el siguiente error:

```
def intercambio(x, y):      # version incorrecta
  x, y = y, x
```

Si llamamos a esta función así:

```
intercambio(a, b)
```

entonces `a` y `x` son dos alias para el mismo valor. Cambiar `x` dentro de `intercambio` hace que `x` se refiera a un valor diferente, pero no tiene efecto en la `a` dentro de `__main__`. Igualmente, cambiar `y` no tiene efecto en `b`.
Esta función se ejecuta sin errores, pero no hace lo que se pretende. Es un ejemplo de error semántico.

> *Como ejercicio, dibuje un diagrama de estados para esta función de manera que se visualice por qué no funciona como debería.*

10.4. Números aleatorios

La gran mayoría de los programas hacen lo mismo cada vez que se ejecutan, esto es, son **determinísticos**. El determinismo generalmente es una buena propiedad, ya que usualmente esperamos que los cálculos produzcan el mismo resultado. Sin embargo, para algunas aplicaciones necesitamos que el computador sea impredecible. Los juegos son un ejemplo inmediato, pero hay más.

Lograr que un programa sea verdaderamente no determinístico no es una tarea fácil, pero hay formas de que parezca no determinístico. Una de ellas es generar números aleatorios y usarlos para determinar la salida de un programa. Python tiene una función primitiva que genera números **pseudoaleatorios**, que, aunque no sean aleatorios desde el punto de vista matemático, sirven para nuestros propósitos.

El módulo `random` contiene una función llamada `random` que retorna un número flotante entre 0.0 y 1.0. Cada vez que se llama a `random`, se obtiene el siguiente número de una serie muy larga. Para ver un ejemplo ejecute el siguiente ciclo:

```
import random

for i in range(10):
  x = random.random()
  print x
```

Para generar un número aleatorio entre 0.0 y un límite superior como `sup`, multiplique `x` por `sup`.

> *Como ejercicio, genere un número aleatorio entre* `inf` *y* `sup`*.*
>
> *Como ejercico adicional genere un número aleatorio* entero *entre* `inf` *y* `sup`*, incluyendo ambos extremos.*

10.5. Lista de números aleatorios

Vamos a generar función que cree una lista de números aleatorios `listaAleatoria`, recibirá un parámetro entero que especifique el número de elementos a generar. Primero, genera una lista de `n` ceros. Luego cada vez que itera en un ciclo for, reemplaza uno de los ceros por un número aleatorio. El valor de retorno es una referencia a la lista construida:

```
def listaAleatoria(n):
  s = [0] * n
  for i in range(n):
    s[i] = random.random()
  return s
```

La probaremos con ocho elementos. Para depurar es una buena idea empezar con pocos datos:

```
>>> listaAleatoria(8)
0.15156642489
0.498048560109
0.810894847068
0.360371157682
0.275119183077
0.328578797631
0.759199803101
0.800367163582
```

Los números generados por `random` deben distribuirse uniformemente, lo que significa que cada valor es igualmente probable.
Si dividimos el rango de valores posibles en "regiones" del mismo tamaño y contamos el número de veces que un valor aleatorio cae en cada región, deberíamos obtener un resultado aproximado en todas las regiones.
Podemos probar esta hipótesis escribiendo un programa que divida el rango en regiones y cuente el número de valores que caen en cada una.

10.6. Conteo

Un enfoque que funciona en problemas como éste es dividir el problema en subproblemas que se puedan resolver con un patrón computacional que ya sepamos.
En este caso, necesitamos recorrer una lista de números y contar el número de veces que un valor cae en un rango dado. Esto parece familiar. En la Sección 8.8, escribimos un programa que recorría una cadena y contaba el números de veces que aparecía una letra determinada.
Entonces podemos copiar el programa viajo para adaptarlo posteriormente a nuestro problema actual. El original es:

```
cont = 0
for c in fruta:
  if c == 'a':
    cont = cont + 1
print cont
```

El primer paso es reemplazar `fruta` con `lista` y `c` por `num`. Esto no cambia el programa, sólo lo hace más legible.

El segundo paso es cambiar la prueba. No queremos buscar letras. Queremos ver si `num` está entre dos valores dados `inf` y `sup`.

```
cont = 0
for num in lista
  if inf < num < sup:
    cont = cont + 1
print cont
```

El último paso consiste en encapsular este código en una función denominada `enRegion`. Los parámetros son la lista y los valores `inf` y `sup`.

```
def enRegion(lista, inf, sup):
  cont = 0
  for num in lista:
    if inf < num < sup:
      cont = cont + 1
  return cont
```

Copiando y modificando un programa existente fuimos capaces de escribir esta función rápidamente y ahorrarnos un buen tiempo de depuración. Este plan de desarrollo se denomina **concordancia de patrones**. Si se encuentra trabajando en un problema que ya ha resuelto antes, reutilice la solución.

10.7. Muchas regiones

Como el número de regiones aumenta, `enRegion` es un poco engorroso. Con dos no esta tan mal:

```
inf = enRegion(a, 0.0, 0.5)
sup = enRegion(a, 0.5, 1)
```

Pero con cuatro:

```
Region1 = enRegion(a, 0.0, 0.25)
Region2 = enRegion(a, 0.25, 0.5)
Region3 = enRegion(a, 0.5, 0.75)
Region4 = enRegion(a, 0.75, 1.0)
```

Hay dos problemas. Uno es que siempre tenemos que crear nuevos nombres de variables para cada resultado. El otro es que tenemos que calcular el rango de cada región.

Primero resolveremos el segundo problema. Si el número de regiones está dado por la variable `numRegiones`, entonces, el ancho de cada región está dado por la expresión `1.0/numRegiones`.
Usaremos un ciclo para calcular el rango de cada región. La variable de ciclo `i` cuenta de 1 a `numRegiones-1`:

```
ancho = 1.0 / numRegiones
for i in range(numRegiones):
  inf = i * ancho
  sup = inf + ancho
  print inf, " hasta ", sup
```

Para calcular el extremo inferior de cada región, multiplicamos la variable de ciclo por el ancho. El extremo superior está a un `ancho` de región de distancia.
Con `numRegiones = 8`, la salida es:

```
0.0 hasta 0.125
0.125 hasta 0.25
0.25 hasta 0.375
0.375 hasta 0.5
0.5 hasta 0.625
0.625 hasta 0.75
0.75 hasta 0.875
0.875 hasta 1.0
```

Usted puede confirmar que cada región tiene el mismo ancho, que no se solapan y que cubren el rango completo de 0.0 a 1.0.
Ahora regresemos al primer problema. Necesitamos una manera de almacenar ocho enteros, usando una variable para indicarlos uno a uno. Usted debe estar pensando "¡una lista!"
Tenemos que crear la lista de regiones fuera del ciclo, porque esto sólo debe ocurrir una vez. Dentro del ciclo, llamaremos a `enRegion` repetidamente y actualizaremos el `i`ésimo elemento de la lista:

```
numRegiones = 8
Regiones = [0] * numRegiones
ancho = 1.0 / numRegiones
for i in range(numRegiones):
  inf = i * ancho
  sup = inf + ancho
  Regiones[i] = enRegion(lista, inf, sup)
print Regiones
```

Con una lista de 1000 valores, este código produce la siguiente lista de conteo:

```
[138, 124, 128, 118, 130, 117, 114, 131]
```

Todos estos valores están muy cerca a 125, que es lo que esperamos. Al menos, están lo suficientemente cerca como para creer que el generador de números pseudoaleatorios está funcionando bien.

> *Como ejercicio, envuelva este código en una función, pruébela con listas más grandes y vea si los números de valores en cada región tienden a emparejarse*

10.8. Una solución en una sola pasada

Aunque funciona, este programa no es tan eficiente como debería. Cada vez que llama a `enRegion`, recorre la lista entera. A medida que el número de regiones incrementa, va a hacer muchos recorridos.
Sería mejor hacer una sola pasada a través de la lista y calcular para cada región el índice de la región en la que cae. Así podemos incrementar el contador apropiado.
En la sección anterior tomamos un índice `i` y lo multiplicamos por el `ancho` para encontrar el extremo inferior de una región. Ahora vamos a encontrar el índice de la región en la que cae.
Como este problema es el inverso del anterior, podemos intentar dividir por `ancho` en vez de multiplicar. ¡Esto funciona!
Como `ancho = 1.0 / numRegiones`, dividir por `ancho` es lo mismo que multiplicar por `numRegiones`. Si multiplicamos un número en el rango 0.0 a 1.0 por `numRegiones`, obtenemos un número en el rango de 0.0 a `numRegiones`. Si redondeamos ese número al entero más cercano por debajo, obtenemos lo que queremos: un índice de región:

```
numRegiones = 8
Regiones = [0] * numRegiones
for i in lista:
  ind = int(i * numRegiones)
  Regiones[ind] = Regiones[ind] + 1
```

Usamos la función `int` para pasar de número de punto flotante a entero.
¿Es posible que este programa produzca un índice que esté fuera de rango (por ser negativo o mayor que `len(Regiones)-1`)?
Una lista como `Regiones` que almacena los conteos del número de valores que hay en cada rango se denomina **histograma**.

> *Como ejercicio, escriba una función llamada* `histograma` *que tome una lista y un número de regiones como parámetros. Debe retornar un histograma del número de regiones dado.*

10.9. Glosario

Tipo inmutable: es un tipo de dato en el que los elementos no pueden ser modificados. Las asignaciones a elementos o segmentos de tipos inmutables causan errores. Las cadenas y las tuplas son inmutables.

Tipo mutable: tipo de dato en el que los elementos pueden ser modificados. Todos los tipos mutables son compuestos. Las listas y los diccionarios son mutables.

Tupla: tipo de dato secuencial similar a la lista, pero inmutable. Las tuplas se pueden usar donde se requiera un tipo inmutable, por ejemplo como llaves de un diccionario.

Asignación de tuplas: una asignación a todos los elementos de una tupla en una sola sentencia. La asignación ocurre en paralelo y no secuencialmente. Es útil para intercambiar valores de variables.

Determinístico: programa que hace lo mismo cada vez que se llama.

Pseudoaleatoria: secuencia de números que parece aleatoria, pero en realidad es el resultado de un cómputo determinístico, bien diseñado.

Histograma: lista de enteros en la que cada elemento cuenta el número de veces que algo sucede.

Correspondencia de patrones: plan de desarrollo de programas que implica identificar un patrón computacional familiar y copiar la solución de un problema similar.

Capítulo 11

Diccionarios

Los tipos compuestos que ha visto hasta ahora (cadenas, listas y tuplas) usan enteros como índices. Si usted intenta usar cualquier otro tipo como índice provocará un error.

Los **diccionarios** son similares a otros tipos compuestos, excepto en que pueden usar como índice cualquier tipo inmutable. A modo de ejemplo, crearemos un diccionario que traduzca palabras inglesas al español. En este diccionario, los índices son cadenas (`strings`).

Una forma de crear un diccionario es empezar con el diccionario vacío y añadir elementos. El diccionario vacío se expresa como {}:

```
>>> ing_a_esp = {}
>>> ing_a_esp['one'] = 'uno'
>>> ing_a_esp['two'] = 'dos'
```

La primera asignación crea un diccionario llamado `ing_a_esp`; las otras asignaciones añaden nuevos elementos al diccionario. Podemos desplegar el valor actual del diccionario del modo habitual:

```
>>> print ing_a_esp
{'one': 'uno', 'two': 'dos'}
```

Los elementos de un diccionario aparecen en una lista separada por comas. Cada entrada contiene un índice y un valor separado por dos puntos (:). En un diccionario, los índices se llaman **claves**, por eso los elementos se llaman **pares clave-valor**.

Otra forma de crear un diccionario es dando una lista de pares clave-valor con la misma sintaxis que la salida del ejemplo anterior:

```
>>> ing_a_esp={'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Si volvemos a imprimir el valor de `ing_a_esp`, nos llevamos una sorpresa:

```
>>> print ing_a_esp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

¡Los pares clave-valor no están en orden! Afortunadamente, no necesitamos preocuparnos por el orden, ya que los elementos de un diccionario nunca se indexan con índices enteros. En lugar de eso, usamos las claves para buscar los valores correspondientes:

```
>>> print ing_a_esp['two']
'dos'
```

La clave `'two'` nos da el valor `'dos'` aunque aparezca en el tercer par clave-valor.

11.1. Operaciones sobre diccionarios

La sentencia `del` elimina un par clave-valor de un diccionario. Por ejemplo, el diccionario siguiente contiene los nombres de varias frutas y el número de esas frutas en un almacén:

```
>>> inventario = {'manzanas': 430, 'bananas': 312,
       'naranjas': 525,   'peras': 217}
>>> print inventario
{'naranjas': 525, 'manzanas': 430, 'peras': 217,
 'bananas': 312}
```

Si alguien compra todas las peras, podemos eliminar la entrada del diccionario:

```
>>> del inventario['peras']
>>> print inventario
{'naranjas': 525, 'manzanas': 430, 'bananas': 312}
```

O si esperamos recibir más peras pronto, podemos simplemente cambiar el inventario asociado con las peras:

```
>>> inventario['peras'] = 0
>>> print inventario
{'naranjas': 525, 'manzanas': 430, 'peras': 0,
 'bananas': 312}
```

La función `len` también funciona con diccionarios; devuelve el número de pares clave-valor:

```
>>> len(inventario)
4
```

11.2. Métodos del diccionario

Un **método** es similar a una función, acepta parámetros y devuelve un valor, pero la sintaxis es diferente. Por ejemplo, el método `keys` acepta un diccionario y devuelve una lista con las claves que aparecen, pero en lugar de la sintaxis de llamado de función `keys(ing_a_esp)`, usamos la sintaxis para un método `ing_a_esp.keys()`.

```
>>> ing_a_esp.keys()
['one', 'three', 'two']
```

Esta forma de notación punto especifica el nombre de la función, `keys`, y el nombre del objeto al que se va a aplicar la función, `ing_a_esp`. Los paréntesis indican que este método no admite parámetros.
La llamada a un método se denomina **invocación**; en este caso, diríamos que estamos invocando `keys` sobre el objeto `ing_a_esp`.
El método `values` es similar; devuelve una lista de los valores del diccionario:

```
>>> ing_a_esp.values()
['uno', 'tres', 'dos']
```

El método `items` devuelve ambos, una lista de tuplas con los pares clave-valor del diccionario:

```
>>> ing_a_esp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]
```

La sintaxis nos proporciona información muy útil acerca del tipo de datos. Los corchetes indican que es una lista. Los paréntesis indican que los elementos de la lista son tuplas.
Si un método acepta un argumento, usa la misma sintaxis que una llamada a una función. Por ejemplo, el método `has_key` acepta una clave y devuelve verdadero (1) si la clave aparece en el diccionario:

```
>>> ing_a_esp.has_key('one')
True
>>> ing_a_esp.has_key('deux')
False
```

Si se invoca un método sin especificar un objeto, se genera un error. En este caso, el mensaje de error no es de mucha ayuda:

```
>>> has_key('one')
NameError: has_key
```

11.3. Copiado y alias

Usted debe estar atento a los alias debido a la mutabilidad de los diccionarios. Si dos variables se refieren al mismo objeto los cambios en una afectan a la otra.
Si quiere modificar un diccionario y mantener una copia del original, se puede usar el método `copy`. Por ejemplo, `opuestos` es un diccionario que contiene pares de opuestos:

```
>>> opuestos = {'arriba': 'abajo', 'derecho': 'torcido',
  'verdadero': 'falso'}
>>> alias = opuestos
>>> copia = opuestos.copy()
```

`alias` y `opuestos` se refieren al mismo objeto; `copia` se refiere a una copia nueva del mismo diccionario. Si modificamos `alias`, `opuestos` también resulta cambiado:

```
>>> alias['derecho'] = 'sentado'
>>> opuestos['derecho']
'sentado'
```

Si modificamos `copia`, `opuestos` no varía:

```
>>> copia['derecho'] = 'privilegio'
>>> opuestos['derecho']
'sentado'
```

11.4. Matrices dispersas

En la Sección 9.14 usamos una lista de listas para representar una matriz. Es una buena opción para una matriz en la que la mayoría de los valores es diferente de cero, pero piense en una matriz como ésta:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

La representación de la lista contiene un montón de ceros:

```
>>> matriz = [ [0,0,0,1,0],
           [0,0,0,0,0],
```

```
           [0,2,0,0,0],
           [0,0,0,0,0],
           [0,0,0,3,0] ]
```

Una posible alternativa consiste en usar un diccionario. Como claves, podemos usar tuplas que contengan los números de fila y columna. Ésta es la representación de la misma matriz por medio de un diccionario:

```
>>> matriz = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

Sólo hay tres pares clave-valor, uno para cada elemento de la matriz diferente de cero. Cada clave es una tupla, y cada valor es un entero.
Para acceder a un elemento de la matriz, podemos usar el operador `[]`:

```
>>> matriz[0,3]
1
```

Observe que la sintaxis para la representación por medio del diccionario no es la misma de la representación por medio de la lista anidada. En lugar de dos índices enteros, usamos un índice compuesto que es una tupla de enteros.
Hay un problema. Si apuntamos a un elemento que es cero, se produce un error porque en el diccionario no hay una entrada con esa clave:

```
>>> matriz[1,3]
KeyError: (1, 3)
```

El método `get` soluciona este problema:

```
>>> matriz.get((0,3), 0)
1
```

El primer argumento es la clave; el segundo argumento es el valor que debe devolver `get` en caso de que la clave no esté en el diccionario:

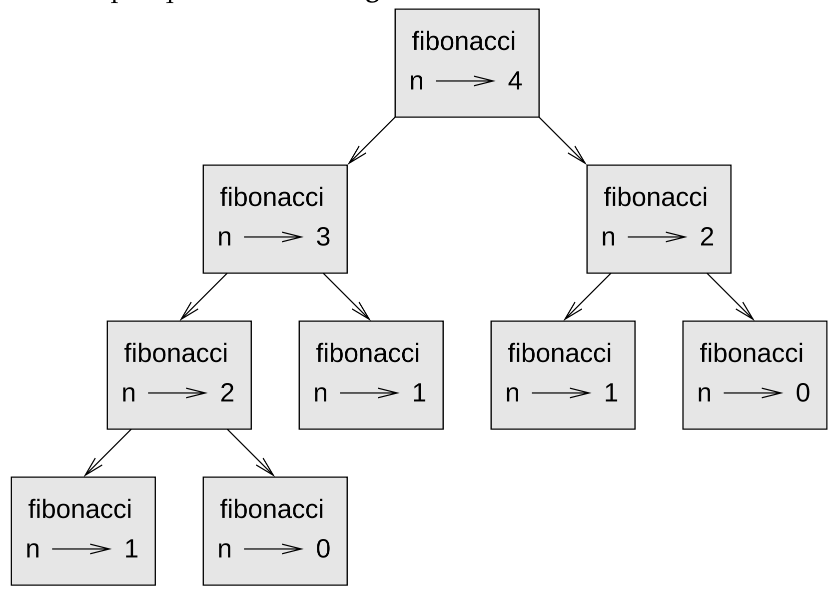
```
>>> matriz.get((1,3), 0)
0
```

`get` mejora sensiblemente la semántica del acceso a una matriz dispersa. ¡Lástima que la sintaxis no sea tan clara!

11.5. Pistas

Si ha jugado con la función `fibonacci` de la Sección 6.7, es posible que haya notado que cuanto más grande es el argumento que recibe, más tiempo le cuesta ejecutarse. De hecho, el tiempo de ejecución aumenta muy rápidamente. En nuestra máquina, `fibonacci(20)` acaba instantáneamente, `fibonacci(30)` tarda más o menos un segundo, y `fibonacci(40)` tarda una eternidad.

Para entender por qué, observe este **gráfico de llamadas** de `fibonacci` con `n=4`:



Un gráfico de llamadas muestra un conjunto de cajas de función con líneas que conectan cada caja con las cajas de las funciones a las que llama. En lo alto del gráfico, `fibonacci` con `n=4` llama a `fibonacci` con `n=3` y `n=2`. A su vez, `fibonacci` con `n=3` llama a `fibonacci` con `n=2` y `n=1`. Y así sucesivamente.

Cuente cuántas veces se llama a `fibonacci(0)` y `fibonacci(1)`. Esta función es una solución ineficiente para el problema, y empeora mucho a medida que crece el argumento.

Una buena solución es llevar un registro de los valores que ya se han calculado almacenándolos en un diccionario. A un valor que ya ha sido calculado y almacenado para un uso posterior se le llama **pista**. Aquí hay una implementación de `fibonacci` con pistas:

```
anteriores = {0:1, 1:1}

def fibonacci(n):
  if anteriores.has_key(n):
```

```
    return anteriores[n]
  else:
    nuevoValor = fibonacci(n-1) + fibonacci(n-2)
    anteriores[n] = nuevoValor
    return nuevoValor
```

El diccionario llamado `anteriores` mantiene un registro de los valores de Fibonacci que ya conocemos. El programa comienza con sólo dos pares: 0 corresponde a 1 y 1 corresponde a 1.
Siempre que se llama a `fibonacci` comprueba si el diccionario contiene el resultado ya calculado. Si está ahí, la función puede devolver el valor inmediatamente sin hacer más llamadas recursivas. Si no, tiene que calcular el nuevo valor. El nuevo valor se añade al diccionario antes de que la función retorne.
Con esta versión de `fibonacci`, nuestra máquina puede calcular `fibonacci(40)` en un abrir y cerrar de ojos. Pero cuando intentamos calcular `fibonacci(50)`, nos encontramos con otro problema:

```
>>> fibonacci(50)
OverflowError: integer addition
```

La respuesta, como se verá en un momento, es 20.365.011.074. El problema es que este número es demasiado grande para caber en un entero de Python. Se **desborda**. Afortunadamente, hay una solución fácil para este problema.

11.6. Enteros largos

Python proporciona un tipo llamado `long int` que puede manejar enteros de cualquier tamaño. Hay dos formas de crear un valor `long int`. Una es escribir un entero con una *L* mayúscula al final:

```
>>> type(1L)
<type ’long int’>
```

La otra es usar la función `long` para convertir un valor en `long int`. `long` acepta cualquier tipo numérico e incluso cadenas de dígitos:

```
>>> long(1)
1L
>>> long(3.9)
3L
>>> long(’57’)
57L
```

Todas las operaciones matemáticas funcionan sobre los datos de tipo `long int`, así que no tenemos que hacer mucho para adaptar `fibonacci`:

```
>>> anterior = {0:1L, 1:1L}
>>> fibonacci(50)
20365011074L
```

Simplemente cambiando el contenido inicial de `anteriores` cambiamos el comportamiento de `fibonacci`. Los primeros dos números de la secuencia son de tipo `long int`, así que todos los números subsiguientes lo serán también.

> *Como ejercicio, modifica* `factorial` *de forma que produzca un* `long int` *como resultado.*

11.7. Contar letras

En el capítulo 8 escribimos una función que contaba el número de apariciones de una letra en una cadena. Una versión más genérica de este problema es crear un histograma de las letras de la cadena, o sea, cuántas veces aparece cada letra.
Ese histograma podría ser útil para comprimir un archivo de texto. Como las diferentes letras aparecen con frecuencias distintas, podemos comprimir un archivo usando códigos cortos para las letras más habituales y códigos más largos para las que aparecen con menor frecuencia.
Los diccionarios facilitan una forma elegante de generar un histograma:

```
>>> cuentaLetras = {}
>>> for letra in "Mississippi":
...   cuentaLetras[letra] = cuentaLetras.get(letra, 0)+1
...
>>> cuentaLetras
{'M': 1, 's': 4, 'p': 2, 'i': 4}
>>>
```

Inicialmente, tenemos un diccionario vacío. Para cada letra de la cadena, buscamos el recuento actual (posiblemente cero) y la incrementamos. Al final, el diccionario contiene pares de letras y sus frecuencias.
Puede ser más atractivo mostrar el histograma en orden alfabético. Podemos hacerlo con los métodos `items` y `sort`:

```
>>> itemsLetras = cuentaLetras.items()
>>> itemsLetras.sort()
>>> print itemsLetras
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Usted ya ha visto el método `items` aplicable a los diccionarios; `sort` es un método aplicable a listas. Hay varios más, como `append`, `extend` y `reverse`. Consulte la documentación de Python para ver los detalles.

11.8. Glosario

Diccionario: es una colección de pares clave-valor que establece una correspondencia entre claves y valores. Las claves pueden ser de cualquier tipo inmutable, los valores pueden ser de cualquier tipo.

Clave: un valor que se usa para buscar una entrada en un diccionario.

Par clave-valor: uno de los elementos de un diccionario, también llamado "asociación".

Método: tipo de función al que se llama con una sintaxis diferente y al que se invoca "sobre" un objeto.

Invocar: llamar a un método.

Pista: almacenamiento temporal de un valor precalculado, para evitar cálculos redundantes.

Desbordamiento: resultado numérico que es demasiado grande para representarse en formato numérico.

Capítulo 12

Archivos y excepciones

Cuando un programa se está ejecutando, sus datos están en la memoria. Cuando un programa termina, o se apaga el computador, los datos de la memoria desaparecen. Para almacenar los datos de forma permanente se deben poner en un **archivo**. Leyendo y escribiendo archivos, los programas pueden intercambiar información entre ellos y generar formatos imprimibles como PDF.

Normalmente los archivos se guardan en un disco duro, disquete o CD-ROM. Cuando hay un gran número de archivos, suelen estar organizados en **directorios** (también llamados "carpetas"). Cada archivo se identifica con un nombre único, o una combinación de nombre de archivo y nombre de directorio.

Trabajar con archivos se parece mucho a hacerlo con libros. Para usar un libro, hay que abrirlo. Cuando uno ha terminado, hay que cerrarlo. Mientras el libro está abierto, se puede escribir en él o leer de él. En cualquier caso, uno sabe en qué lugar del libro se encuentra. Casi siempre se lee un libro según su orden natural, pero también se puede ir saltando de página en página.

Todo esto sirve también para los archivos. Para abrir un archivo, se especifica su nombre y se indica si se desea leer o escribir.

La apertura de un archivo crea un objeto archivo. En este ejemplo, la variable `f` apunta al nuevo objeto archivo.

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

La función open toma dos argumentos: el primero, es el nombre del archivo y el segundo, el modo. El modo `"w"` significa que lo estamos abriendo para escribir.

Si no hay un archivo llamado `test.dat` se creará. Si ya hay uno, el archivo que estamos escribiendo lo reemplazará.

Al imprimir el objeto archivo, vemos el nombre del archivo, el modo y la localización del objeto.
Para escribir datos en el archivo invocamos al método `write` sobre el objeto archivo:

```
>>> f.write("Ya es hora")
>>> f.write("de cerrar el archivo")
```

El cierre del archivo le dice al sistema que hemos terminado de escribir y deja el archivo listo para leer:

```
>>> f.close()
```

Ya podemos abrir el archivo de nuevo, esta vez para lectura, y poner su contenido en una cadena. Esta vez el argumento de modo es `"r"`, para lectura:

```
>>> f = open("test.dat","r")
```

Si intentamos abrir un archivo que no existe, recibimos un mensaje de error:

```
>>> f = open("test.cat","r")
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Como era de esperar, el método `read` lee datos del archivo. Sin argumentos, lee el archivo completo:

```
>>> texto = f.read()
>>> print texto
Ya es horade cerrar el archivo
```

No hay un espacio entre "hora" y "de" porque no escribimos un espacio entre las cadenas. `read` también puede aceptar un argumento que le indica cuántos carácteres leer:

```
>>> f = open("test.dat","r")
>>> print f.read(7)
Ya es h
```

Si no quedan suficientes carácteres en el archivo, `read` devuelve los que haya. Cuando llegamos al final del archivo, `read` devuelve una cadena vacía:

```
>>> print f.read(1000006)
orade cerrar el archivo
>>> print f.read()

>>>
```

La siguiente función copia un archivo, leyendo y escribiendo los carácteres de cincuenta en cincuenta. El primer argumento es el nombre del archivo original; el segundo es el nombre del archivo nuevo:

```
def copiaArchivo(archViejo, archNuevo):
  f1 = open(archViejo, "r")
  f2 = open(archNuevo, "w")
  while True:
    texto = f1.read(50)
    if texto == "":
      break
    f2.write(texto)
  f1.close()
  f2.close()
  return
```

La sentencia `break` es nueva. Su ejecución interrumpe el ciclo; el flujo de la ejecución pasa a la primera sentencia después del while.
En este ejemplo, el ciclo `while` es infinito porque la condición `True` siempre es verdadera. La *única* forma de salir del ciclo es ejecutar `break`, lo que sucede cuando `texto` es una cadena vacía, y esto pasa cuando llegamos al final del archivo.

12.1. Archivos de texto

Un **archivo de texto** contiene carácteres imprimibles y espacios organizados en líneas separadas por carácteres de salto de línea. Como Python está diseñado específicamente para procesar archivos de texto, proporciona métodos que facilitan esta tarea.
Para hacer una demostración, crearemos un archivo de texto con tres líneas de texto separadas por saltos de línea:

```
>>> f = open("test.dat","w")
>>> f.write("línea uno\nlínea dos\nlínea tres\n")
>>> f.close()
```

El método `readline` lee todos los carácteres hasta, e incluyendo, el siguiente salto de línea:

```
>>> f = open("test.dat","r")
>>> print f.readline()
línea uno

>>>
```

`readlines` devuelve todas las líneas que queden como una lista de cadenas:

```
>>> print f.readlines()
['línea dos\012', 'línea tres\012']
```

En este caso, la salida está en forma de lista, lo que significa que las cadenas aparecen con comillas y el carácter de salto de línea aparece como la secuencia de escape `012`.
Al final del archivo, `readline` devuelve una cadena vacía y `readlines` devuelve una lista vacía:

```
>>> print f.readline()

>>> print f.readlines()
[]
```

Lo que sigue es un ejemplo de un programa de proceso de líneas. `filtraArchivo` hace una copia de `archViejo`, omitiendo las líneas que comienzan por #:

```
def filtraArchivo(archViejo, archNuevo):
  f1 = open(archViejo, "r")
  f2 = open(archNuevo, "w")
  while 1:
    texto = f1.readline()
    if texto == "":
      break
    if texto[0] == '#':
      continue
    f2.write(texto)
  f1.close()
  f2.close()
  return
```

La sentencia `continue` termina la iteración actual del ciclo, pero sigue haciendo las que le faltan. El flujo de ejecución pasa al principio del ciclo, comprueba la condición y continúa normalmemente.
Así, si `texto` es una cadena vacía, el ciclo termina. Si el primer caracter de `texto` es una almohadilla `(#)`, el flujo de ejecución va al principio del ciclo. Sólo si ambas condiciones fallan copiamos `texto` en el archivo nuevo.

12.2. Escribir variables

El argumento de `write` debe ser una cadena, así que si queremos poner otros valores en un archivo, tenemos que convertirlos previamente en cadenas. La forma más fácil de hacerlo es con la función `str`:

```
>>> x = 52
>>> f.write (str(x))
```

Una alternativa es usar el **operador de formato** `%`. Cuando aplica a enteros, `%` es el operador de módulo. Pero cuando el primer operando es una cadena, `%` es el operador de formato.
El primer operando es la **cadena de formato**, y el segundo, una tupla de expresiones. El resultado es una cadena que contiene los valores de las expresiones, formateados de acuerdo con la cadena de formato.
A modo de ejemplo simple, la **secuencia de formato** `"%d"` significa que la primera expresión de la tupla debería formatearse como un entero. Aquí la letra *d* quiere decir ”decimal”:

```
>>> motos = 52
>>> "%d" % motos
'52'
```

El resultado es la cadena `'52'`, que no debe confundirse con el valor entero `52`.
Una secuencia de formato puede aparecer en cualquier lugar de la cadena de formato, de modo que podemos incrustar un valor en una frase:

```
>>> motos = 52
>>> "En julio vendimos %d motos." % motos
'En julio vendimos 52 motos.'
```

La secuencia de formato `"%f"` formatea el siguiente elemento de la tupla como un número en punto flotante, y `"%s"` formatea el siguiente elemento como una cadena:

```
>>> "En %d días ganamos %f millones de %s." % (4,1.2,'pesos')
'En 34 días ganamos 1.200000 millones de pesos.'
```

Por defecto, el formato de punto flotante imprime seis decimales.
El número de expresiones en la tupla tiene que coincidir con el número de secuencias de formato de la cadena. Igualmente, los tipos de las expresiones deben coincidir con las secuencias de formato:

```
>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dólares'
TypeError: illegal argument type for built-in operation
```

En el primer ejemplo no hay suficientes expresiones; en el segundo, la expresión es de un tipo incorrecto.
Para tener más control sobre el formato de los números, podemos detallar el número de dígitos como parte de la secuencia de formato:

```
>>> "%6d" % 62
'    62'
>>> "%12f" % 6.1
'    6.100000'
```

El número tras el signo de porcentaje es el número mínimo de espacios que ocupará el número. Si el valor necesita menos dígitos, se añaden espacios en blanco delante del número. Si el número de espacios es negativo, se añaden los espacios tras el número:

```
>>> "%-6d" % 62
'62    '
```

También podemos especificar el número de decimales para los números en coma flotante:

```
>>> "%12.2f" % 6.1
'        6.10'
```

En este ejemplo, el resultado ocupa doce espacios e incluye dos dígitos tras la coma. Este formato es útil para imprimir cantidades de dinero con las comas alineadas.
Imagine, por ejemplo, un diccionario que contiene los nombres de los estudiantes como clave y las tarifas horarias como valores. He aquí una función que imprime el contenido del diccionario como de un informe formateado:

```
def informe (tarifas) :
  estudiantes = tarifas.keys()
  estudiantes.sort()
  for estudiante in estudiantes :
    print "%-20s %12.02f"%(estudiante, tarifas[estudiante])
```

Para probar la función, crearemos un pequeño diccionario e imprimiremos el contenido:

```
>>> tarifas = {'maría': 6.23, 'josé': 5.45, 'jesús': 4.25}
>>> informe (tarifas)
josé                        5.45
jesús                       4.25
maría                       6.23
```

Controlando el ancho de cada valor nos aseguramos de que las columnas van a quedar alineadas, siempre que los nombre tengan menos de veintiún carácteres y las tarifas sean menos de mil millones la hora.

12.3. Directorios

Cuando se crea un archivo nuevo abriéndolo y escribiendo, este va a quedar en el directorio en uso (aquél en el que se estuviese al ejecutar el programa). Del mismo modo, cuando se abre un archivo para leerlo, Python lo busca en el directorio en uso.
Si usted quiere abrir un archivo de cualquier otro sitio, tiene que especificar la **ruta** del archivo, que es el nombre del directorio (o carpeta) donde se encuentra este:

```
>>>   f = open("/usr/share/dict/words","r")
>>>   print f.readline()
Aarhus
```

Este ejemplo abre un archivo denominado `words`, que se encuentra en un directorio llamado `dict`, que está en `share`, en `usr`, que está en el directorio de nivel superior del sistema, llamado `/`.
No se puede usar `/` como parte del nombre de un archivo; está reservado como delimitador entre nombres de archivo y directorios.
El archivo `/usr/share/dict/words` contiene una lista de palabras en orden alfabético, la primera de las cuales es el nombre de una universidad danesa.

12.4. Encurtido

Para poner valores en un archivo, se deben convertir a cadenas. Usted ya ha visto cómo hacerlo con `str`:

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

El problema es que cuando se vuelve a leer el valor, se obtiene una cadena. Se ha perdido la información del tipo de dato original. En realidad, no se puede distinguir dónde termina un valor y dónde comienza el siguiente:

```
>>>   f.readline()
'12.3[1, 2, 3]'
```

La solución es el **encurtido**, llamado así porque "encurte" estructuras de datos. El módulo `pickle` contiene las órdenes necesarias. Para usarlo, se importa `pickle` y luego se abre el archivo de la forma habitual:

```
>>> import pickle
>>> f = open("test.pck","w")
```

Para almacenar una estructura de datos, se usa el método `dump` y luego se cierra el archivo de la forma habitual:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Ahora podemos abrir el archivo para leer y cargar las estructuras de datos que volcamos ahí:

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x
12.3
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type 'list'>
```

Cada vez que invocamos `load` obtenemos un valor del archivo completo con su tipo original.

12.5. Excepciones

Siempre que ocurre un error en tiempo de ejecución, se crea una **excepción**. Normalmente el programa se para y Python presenta un mensaje de error.
Por ejemplo, la división por cero crea una excepción:

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

Un elemento no existente en una lista hace lo mismo:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

O el acceso a una clave que no está en el diccionario:

```
>>> b = {}
>>> print b['qué']
KeyError: qué
```

En cada caso, el mensaje de error tiene dos partes: el tipo de error antes de los dos puntos y detalles sobre el error después de los dos puntos. Normalmente, Python también imprime una traza de dónde se encontraba el programa, pero la hemos omitido en los ejemplos.

A veces queremos realizar una operación que podría provocar una excepción, pero no queremos que se pare el programa. Podemos **manejar** la excepción usando las sentencias `try` y `except`.

Por ejemplo, podemos preguntar al usuario por el nombre de un archivo y luego intentar abrirlo. Si el archivo no existe, no queremos que el programa se aborte; queremos manejar la excepción.

```
nombreArch = raw_input('Introduce un nombre de archivo: ')
try:
  f = open (nombreArch, "r")
except:
  print 'No hay ningún archivo que se llame', nombreArch
```

La sentencia `try` ejecuta las sentencias del primer bloque. Si no se produce ninguna excepción, pasa por alto la sentencia `except`. Si ocurre cualquier excepción, ejecuta las sentencias de la rama `except` y después continúa.

Podemos encapsular esta capacidad en una función: `existe`, que acepta un nombre de archivo y devuelve verdadero si el archivo existe y falso si no:

```
def existe(nombreArch):
  try:
    f = open(nombreArch)
    f.close()
    return True
  except:
    return False
```

Se pueden usar múltiples bloques `except` para manejar diferentes tipos de excepciones. El *Manual de Referencia de Python* contiene los detalles.

Si su programa detecta una condición de error, se puede lanzar (**raise** en inglés) una excepción. Aquí hay un ejemplo que acepta una entrada del usuario y comprueba si es 17. Suponiendo que 17 no es una entrada válida por cualquier razón, lanzamos una excepción.

```
def tomaNumero () :
  # ¡Recuerda, los acentos están prohibidos
  x = input ('Elige un número: ')
  # en los nombres de funciones y variables!
  if x == 17 :
```

```
    raise 'ErrorNúmeroMalo', '17 es un mal número'
  return x
```

La sentencia `raise` acepta dos argumentos: el tipo de excepción e información específica acerca del error. `ErrorNúmeroMalo` es un nuevo tipo de excepción que hemos inventado para esta aplicación.
Si la función llamada `tomaNumero` maneja el error, el programa puede continuar; en caso contrario, Python imprime el mensaje de error y sale:

```
>>> tomaNumero ()
Elige un número: 17
ErrorNúmeroMalo: 17 es un mal número
```

El mensaje de error incluye el tipo de excepción y la información adicional proporcionada.

> *Como ejercicio, escribe una función que use* `tomaNumero` *para leer un número del teclado y que maneje la excepción* `ErrorNumeroMalo`.

12.6. Glosario

Archivo: entidad con nombre, normalmente almacenada en un disco duro, disquete o CD-ROM, que contiene una secuencia de carácteres.

Directorio: colección de archivos, con nombre, también llamado carpeta.

Ruta: secuencia de nombres de directorio que especifica la localización exacta de un archivo.

Archivo de texto: un archivo que contiene carácteres imprimibles organizados en líneas separadas por carácteres de salto de línea.

Sentencia break: es una sentencia que provoca que el flujo de ejecución salga de un ciclo.

Sentencia continue: sentencia que provoca que termine la iteración actual de un ciclo. El flujo de la ejecución va al principio del ciclo, evalúa la condición, y procede en consecuencia.

Operador de formato: el operador `%` toma una cadena de formato y una tupla de expresiones y entrega una cadena que incluye las expresiones, formateadas de acuerdo con la cadena de formato.

Cadena de formato: una cadena que contiene carácteres imprimibles y secuencias de formato que indican cómo dar formato a valores.

Secuencia de formato: secuencia de carácteres que comienza con `%` e indica cómo dar formato a un valor.

Encurtir: escribir el valor de un dato en un archivo junto con la información sobre su tipo de forma que pueda ser reconstituido más tarde.

Excepción: error que ocurre en tiempo de ejecución.

Manejar: impedir que una excepción detenga un programa utilizando las sentencias `except` y `try`.

Capítulo 13

Clases y objetos

13.1. Tipos compuestos definidos por el usuario

Una vez utilizados algunos de los tipos internos de Python, estamos listos para crear un tipo definido por el usuario: el `Punto`.

Piense en el concepto de un punto matemático. En dos dimensiones, un punto tiene dos números (coordenadas) que se tratan colectivamente como un solo objeto. En notación matemática, los puntos suelen escribirse entre paréntesis con una coma separando las coordenadas. Por ejemplo, $(0, 0)$ representa el origen, y (x, y) representa el punto x unidades a la derecha e y unidades hacia arriba desde el origen.

Una forma natural de representar un punto en Python es con dos valores en punto flotante. La cuestión es, entonces, cómo agrupar esos dos valores en un objeto compuesto. La solución rápida y burda es utilizar una lista o tupla, y para algunas aplicaciones esa podría ser la mejor opción.

Una alternativa es que el usuario defina un nuevo tipo de dato compuesto, también llamado una **clase**. Esta aproximación exige un poco más de esfuerzo, pero tiene algunas ventajas que pronto se harán evidentes.

Una definición de clase se parece a esto:

```
class Punto:
  pass
```

Las definiciones de clase pueden aparecer en cualquier lugar de un programa, pero normalmente están al principio (tras las sentencias `import`). Las reglas sintácticas de la definición de clases son las mismas que para las otras sentencias compuestas. (ver la Sección 5.5).

Esta definición crea una nueva clase llamada `Punto`. La sentencia **pass** no tiene efectos; sólo es necesaria porque una sentencia compuesta debe tener algo en su cuerpo.
Al crear la clase `Punto` hemos creado un nuevo tipo, que también se llama `Punto`. Los miembros de este tipo se llaman **instancias** del tipo u **objetos**. La creación de una nueva instancia se llama **instanciación**. Para instanciar un objeto `Punto` ejecutamos una función que se llama (probablemente usted ha adivinado) `Punto`:

```
limpio = Punto()
```

A la variable `limpio` se le asigna una referencia a un nuevo objeto `Punto`. A una función como `Punto` que crea un objeto nuevo se le llama **constructor**.

13.2. Atributos

Podemos añadir nuevos datos a una instancia utilizando la notación de punto:

```
>>> limpio.x = 3.0
>>> limpio.y = 4.0
```

Esta sintaxis es similar a la usada para seleccionar una variable de un módulo, como `math.pi` o `string.uppercase`. En este caso, sin embargo, estamos seleccionando un dato de una instancia. Estos datos con nombre se denominan **atributos**.
El diagrama de estados que sigue muestra el resultado de esas asignaciones:

La variable `limpio` apunta a un objeto Punto, que contiene dos atributos. Cada atributo apunta a un número en punto flotante.
Podemos leer el valor de un atributo utilizando la misma sintaxis:

```
>>> print limpio.y
4.0
>>> x = limpio.x
>>> print x
3.0
```

La expresión `limpio.x` significa, "ve al objeto al que apunta `limpio` y toma el valor de `x`". En este caso, asignamos ese valor a una variable llamada `x`. No hay conflicto entre la variable `x` y el atributo `x`. El propósito de la notación de punto es identificar de forma inequívoca a qué variable se refiere el programador.
Se puede usar la notación de punto como parte de cualquier expresión. Así, las sentencias que siguen son correctas:

```
print '(' + str(limpio.x) + ', ' + str(limpio.y) + ')'
distanciaAlCuadrado = limpio.x * limpio.x +
                      limpio.y * limpio.y
```

La primera línea presenta `(3.0, 4.0)`; la segunda línea calcula el valor 25.0.
Usted puede estar tentado a imprimir el propio valor de `limpio`:

```
>>> print limpio
<__main__.Point instance at 80f8e70>
```

El resultado indica que `limpio` es una instancia de la clase `Punto` que se definió en `__main__`. `80f8e70` es el identificador único de este objeto, escrito en hexadecimal. Probablemente ésta no es la manera más clara de mostrar un objeto `Punto`. En breve veremos cómo cambiar esto.

> *Como ejercicio, cree e imprima un objeto `Punto` y luego use `id` para imprimir el identificador único del objeto. Traduzca el número hexadecimal a decimal y asegúrese de que coincidan.*

13.3. Instancias como parámetro

Se puede pasar una instancia como parámetro de la forma habitual. Por ejemplo:

```
def imprimePunto(p):
  print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

`imprimePunto` acepta un punto como argumento y lo muestra en el formato estándar de la matemática. Si se llama a `imprimePunto(limpio)`, el resultado es `(3.0, 4.0)`.

> *Como ejercicio, reescriba la función `distancia` de la Sección 6.2 de forma que acepte dos `Puntos` como parámetros en lugar de cuatro números.*

13.4. Mismidad

El significado de la palabra “mismo” parece totalmente claro hasta que uno se detiene a pensarlo un poco y se da cuenta de que hay algo más de lo que se supone comúnmente.
Por ejemplo, si alguien dice “Pepe y yo tenemos la misma moto”, lo que quiere decir es que su moto y la de Pepe son de la misma marca y modelo, pero que son dos motos distintas. Si dice “Pepe y yo tenemos la misma madre”, quiere decir que

su madre y la de Pepe son la misma persona[1]. Así que la idea de "identidad" es diferente según el contexto.

Cuando uno habla de objetos, hay una ambigüedad parecida. Por ejemplo, si dos `Puntos` son el mismo, ¿significa que contienen los mismos datos (coordenadas) o que son de verdad el mismo objeto?

Para averiguar si dos referencias se refieren al mismo objeto, se utiliza el operador ==. Por ejemplo:

```
>>> p1 = Punto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Punto()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 == p2
False
```

Aunque `p1` y `p2` contienen las mismas coordenadas, no son el mismo objeto. Si asignamos `p1` a `p2`, las dos variables son alias del mismo objeto:

```
>>> p2 = p1
>>> p1 == p2
True
```

Este tipo de igualdad se llama **igualdad superficial**, porque sólo compara las referencias, pero no el contenido de los objetos.

Para comparar los contenidos de los objetos (**igualdad profunda**) podemos escribir una función llamada `mismoPunto`:

```
def mismoPunto(p1, p2) :
  return (p1.x == p2.x) and (p1.y == p2.y)
```

Si ahora creamos dos objetos diferentes que contienen los mismos datos podremos usar `mismoPunto` para averiguar si representan el mismo punto:

```
>>> p1 = Punto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Punto()
>>> p2.x = 3
```

[1]No todas las lenguas tienen el mismo problema. Por ejemplo, el alemán tiene palabras diferentes para los diferentes tipos de identidad. "Misma moto" en este contexto sería "gleiche Motorrad" y "misma madre" sería "selbe Mutter".

```
>>> p2.y = 4
>>> mismoPunto(p1, p2)
True
```

Por supuesto, si las dos variables apuntan al mismo objeto `mismoPunto` devuelve verdadero.

13.5. Rectángulos

13.8
Digamos que queremos una clase que represente un rectángulo. La pregunta es, ¿qué información tenemos que proporcionar para definir un rectángulo? Para simplificar las cosas, supongamos que el rectángulo está orientado vertical u horizontalmente, nunca en diagonal.
Tenemos varias posibilidades: podemos señalar el centro del rectángulo (dos coordenadas) y su tamaño (anchura y altura); o podemos señalar una de las esquinas y el tamaño; o podemos señalar dos esquinas opuestas. Un modo convencional es señalar la esquina superior izquierda del rectángulo y el tamaño.
De nuevo, definiremos una nueva clase:

```
# ¡Prohibidos los acentos fuera de las cadenas!
class Rectangulo:
  pass
```

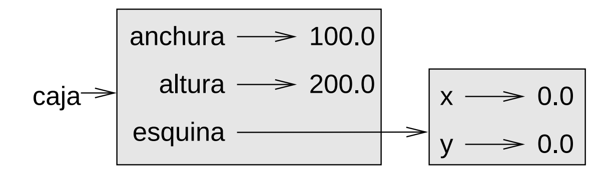
Y la instanciaremos:

```
caja = Rectangulo()
caja.anchura = 100.0
caja.altura = 200.0
```

Este código crea un nuevo objeto `Rectangulo` con dos atributos flotantes. ¡Para señalar la esquina superior izquierda podemos incrustar un objeto dentro de otro!

```
caja.esquina = Punto()
caja.esquina.x = 0.0;
caja.esquina.y = 0.0;
```

El operador punto compone. La expresión `caja.esquina.x` significa ”ve al objeto al que se refiere `caja` y selecciona el atributo llamado `esquina`; entonces ve a ese objeto y selecciona el atributo llamado `x`”.
La figura muestra el estado de este objeto:



13.6. Instancias como valores de retorno

Las funciones pueden devolver instancias. Por ejemplo, `encuentraCentro` acepta un `Rectangulo` como argumento y devuelve un `Punto` que contiene las coordenadas del centro del `Rectangulo`:

```
def encuentraCentro(caja):
  p = Punto()
  p.x = caja.esquina.x + caja.anchura/2.0
  p.y = caja.esquina.y + caja.altura/2.0
  return p
```

Para llamar a esta función, se pasa `caja` como argumento y se asigna el resultado a una variable:

```
>>> centro = encuentraCentro(caja)
>>> imprimePunto(centro)
(50.0, 100.0)
```

13.7. Los objetos son mutables

Podemos cambiar el estado de un objeto efectuando una asignación sobre uno de sus atributos. Por ejemplo, para cambiar el tamaño de un rectángulo sin cambiar su posición, podemos cambiar los valores de `anchura` y `altura`:

```
caja.anchura = caja.anchura + 50
caja.altura = caja.altura + 100
```

Podemos encapsular este código en un método y generalizarlo para agrandar el rectángulo en cualquier cantidad:

```
def agrandarRect(caja, danchura, daltura) :
  caja.anchura = caja.anchura + danchura
  caja.altura = caja.altura + daltura
```

Las variables `danchura` y `daltura` indican cuánto debe agrandarse el rectángulo en cada dirección. Invocar este método tiene el efecto de modificar el `Rectangulo` que se pasa como argumento.

Por ejemplo, podemos crear un nuevo `Rectangulo` llamado `b` y pasárselo a la función `agrandarRect`:

```
>>> b = Rectangulo()
>>> b.anchura = 100.0
>>> b.altura = 200.0
>>> b.esquina = Punto()
>>> b.esquina.x = 0.0;
>>> b.esquina.y = 0.0;
>>> agrandarRect(b, 50, 100)
```

Mientras `agrandarRect` se está ejecutando, el parámetro `caja` es un alias de `b`. Cualquier cambio que se hagas a `caja` afectará también a `b`.

> *A modo de ejercicio, escriba una función llamada* `mueveRect` *que tome un* `Rectangulo` *y dos parámetros llamados* `dx` *y* `dy`*. Tiene que cambiar la posición del rectángulo añadiendo en la* `esquina`*:* `dx` *a la coordenada* `x` *y* `dy` *a la coordenada* `y`*.*

13.8. Copiado

El uso de un alias puede hacer que un programa sea difícil de leer, porque los cambios hechos en un lugar pueden tener efectos inesperados en otro lugar. Es difícil estar al tanto de todas las variables que pueden apuntar a un objeto dado.

Copiar un objeto es, muchas veces, una alternativa a la creación de un alias. El módulo `copy` contiene una función llamada `copy` que puede duplicar cualquier objeto:

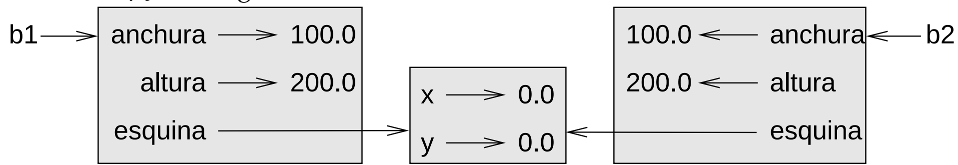
```
>>> import copy
>>> p1 = Punto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
False
>>> mismoPunto(p1, p2)
True
```

Una vez que hemos importado el módulo `copy`, podemos usar el método `copy` para hacer un nuevo `Punto`. `p1` y `p2` no son el mismo punto, pero contienen los mismos datos.

Para copiar un objeto simple como un `Punto`, que no contiene objetos incrustados, `copy` es suficiente. Esto se llama **copiado superficial**.

Para algo como un `Rectangulo`, que contiene una referencia a un `Punto`, `copy` no lo hace del todo bien. Copia la referencia al objeto `Punto`, de modo que tanto el `Rectangulo` viejo como el nuevo apuntan a un único `Punto`.

Si creamos una caja, `b1`, de la forma habitual y entonces hacemos una copia, `b2`, usando `copy`, el diagrama de estados resultante se ve así:



Es casi seguro que esto no es lo que queremos. En este caso, la invocación de `agrandaRect` sobre uno de los `Rectangulo`s no afectaría al otro, ¡pero la invocación de `mueveRect` sobre cualquiera afectaría a ambos! Este comportamiento es confuso y propicia los errores.

Afortunadamente, el módulo `copy` contiene un método llamado `deepcopy` que copia no sólo el objeto, sino también cualesquiera objetos incrustados en él. No lo sorprenderá saber que esta operación se llama **copia profunda** (deep copy).

```
>>> b2 = copy.deepcopy(b1)
```

Ahora `b1` y `b2` son objetos totalmente independientes. Podemos usar `deepcopy` para reescribir `agrandaRect` de modo que en lugar de modificar un `Rectangulo` existente, cree un nuevo `Rectangulo` que tiene la misma localización que el viejo pero nuevas dimensiones:

```
def agrandaRect(caja, danchura, daltura) :
  import copy
  nuevaCaja = copy.deepcopy(caja)
  nuevaCaja.anchura = nuevaCaja.anchura + danchura
  nuevaCaja.altura = nuevaCaja.altura + daltura
  return nuevaCaja
```

> *Como ejercicio, reescriba* `mueveRect` *de modo que cree y devuelva un nuevo* `Rectangulo` *en lugar de modificar el viejo.*

13.9. Glosario

Clase: tipo compuesto definido por el usuario. También se puede pensar en una clase como una plantilla para los objetos que son instancias de la misma.

Instanciar: Crear una instancia de una clase.

Instancia: objeto que pertenece a una clase.

Objeto: tipo de dato compuesto que suele usarse para representar una cosa o concepto del mundo real.

Constructor: método usado para crear nuevos objetos.

Atributo: uno de los elementos de datos con nombre que constituyen una instancia.

Igualdad superficial: igualdad de referencias, o dos referencias que apuntan al mismo objeto.

Igualdad profunda: igualdad de valores, o dos referencias que apuntan a objetos que tienen el mismo valor.

Copia superficial: copiar el contenido de un objeto, incluyendo cualquier referencia a objetos incrustados; implementada por la función `copy` del módulo `copy`.

Copia profunda: copiar el contenido de un objeto así como cualesquiera objetos incrustados, y los incrustados en estos, y así sucesivamente. Está implementada en la función `deepcopy` del módulo `copy`.

Capítulo 14

Clases y funciones

14.1. Hora

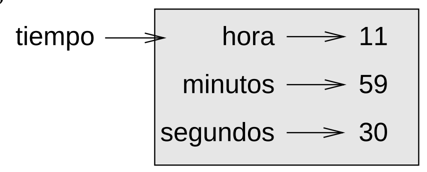
Como otro ejemplo de tipo de dato definido por el usuario definiremos una clase llamada `Hora`:

```
class Hora:
  pass
```

Ahora podemos crear un nuevo objeto `Hora` y asignarle atributos para las horas, minutos y segundos:

```
tiempo = Hora()
tiempo.hora = 11
tiempo.minutos = 59
tiempo.segundos = 30
```

El diagrama para el objeto `Hora` luce así:



> *Como ejercicio, escriba una función* `imprimirHora` *que reciba un objeto* `Hora` *como argumento y lo imprima de la forma* `horas:minutos:segundos`*.*

> *Escriba una función booleana* `despues` *que reciba dos objetos* `Hora`*,* `t1` *y* `t2` *como argumentos, y retorne cierto si* `t1` *va después de* `t2` *cronológicamente y falso en caso contrario.*

14.2. Funciones puras

En las siguientes secciones escribiremos dos versiones de una función denominada `sumarHoras`, que calcule la suma de dos `Horas`. Esto demostrará dos clases de funciones: las puras y los modificadores.
La siguiente es una versión de `sumarHoras`:

```
def sumarHoras(t1, t2):
  sum = Hora()
  sum.hora = t1.hora + t2.hora
  sum.minutos = t1.minutos + t2.minutos
  sum.segundos = t1.segundos + t2.segundos
  return sum
```

La función crea un nuevo objeto `Hora`, inicializa sus atributos y retorna una referencia hacia el nuevo objeto. Esto se denomina **función pura**, porque no modifica ninguno de los objetos que se le pasaron como parámetro ni tiene efectos laterales, como desplegar un valor o leer entrada del usuario.
Aquí hay un ejemplo de uso de esta función. Crearemos dos objetos `Hora`: `horaPan`, que contiene el tiempo que le toma a un panadero hacer pan y `horaActual`, que contiene la hora actual. Luego usaremos `sumarHoras` para averiguar a qué hora estará listo el pan. Si no ha terminado la función `imprimirHora` aún, adelántese a la Sección 15.2 antes de intentar esto:

```
>>> horaActual = Hora()
>>> horaActual.hora = 9
>>> horaActual.minutos = 14
>>> horaActual.segundos =  30

>>> horaPan = Hora()
>>> horaPan.hora =  3
>>> horaPan.minutos =  35
>>> horaPan.segundos =  0

>>> horaComer = sumarHoras(horaActual, horaPan)
>>> imprimirHora(horaComer)
```

La salida de este programa es `12:49:30`, que está correcta. Por otro lado, hay casos en los que no funciona bien. ¿Puede pensar en uno?
El problema radica en que esta función no considera los casos donde el número de segundos o minutos suman más de sesenta. Cuando eso ocurre tenemos que "acarrear" los segundos extra a la columna de minutos. También puede pasar lo mismo con los minutos.
Aquí hay una versión correcta:

```
def sumarHoras(t1, t2):
  sum = Hora()
  sum.hora = t1.hora + t2.hora
  sum.minutos = t1.minutos + t2.minutos
  sum.segundos = t1.segundos + t2.segundos

  if sum.segundos >= 60:
    sum.segundos = sum.segundos - 60
    sum.minutos = sum.minutos + 1

  if sum.minutos >= 60:
    sum.minutos = sum.minutos - 60
    sum.hora = sum.hora + 1

  return sum
```

Aunque ahora ha quedado correcta, ha empezado a agrandarse. Más adelante sugeriremos un enfoque alternativo que produce un código más corto.

14.3. Modificadoras

A veces es deseable que una función modifique uno o varios de los objetos que recibe como parámetros. Usualmente, el código que hace el llamado a la función conserva una referencia a los objetos que está pasando, así que cualquier cambio que la función les haga será evidenciado por dicho código. Este tipo de funciones se denominan **modificadoras**.
`incrementar`, que agrega un número de segundos a un objeto `Hora`, se escribiría más naturalmente como función modificadora. Un primer acercamiento a la función luciría así:

```
def incrementar(h, segundos):
  h.segundos = h.segundos + segundos

  if h.segundos >= 60:
    h.segundos = h.segundos - 60
    h.minutos = h.minutos + 1

  if h.minuto >= 60:
    h.minutos = h.minutos - 60
    h.hora = h.hora + 1

  return h
```

La primera línea ejecuta la operación básica, las siguientes consideran los casos especiales que ya habíamos visto.
¿Es correcta esta función? ¿Que pasa si el parámetro `segundos` es mucho más grande que sesenta? En ese caso, no sólo es suficiente añadir uno, tenemos que sumar de uno en uno hasta que `segundos` sea menor que sesenta. Una solución consiste en reemplazar las sentencias `if` por sentencias `while`:

```
def incrementar(hora, segundos):

  while hora.segundos >= 60:
    hora.segundos = hora.segundos - 60
    hora.minutos = hora.minutos + 1

  while hora.minutos >= 60:
    hora.minutos = hora.minutos - 60
    hora.hora = hora.hora + 1

  return hora

  hora.segundos = hora.segundos + segundos
```

Ahora, la función sí es correcta, aunque no sigue el proceso más eficiente.

> *Como ejercicio, reescriba la función de forma que no contenga ciclos y siga siendo correcta.*
>
> *Reescriba `incrementar` como una función pura, y escriba llamados a funciones de las dos versiones.*

14.4. ¿Cual es el mejor estilo?

Todo lo que puede hacerse con modificadoras también se puede hacer con funciones puras. De hecho, algunos lenguajes de programación sólo permiten funciones puras. La evidencia apoya la tesis de que los programas que usan solamente funciones puras se desarrollan más rápido y son menos propensos a errores que los programas que usan modificadoras. Sin embargo, las funciones modificadoras, a menudo, son convenientes y, a menudo, los programas funcionales puros son menos eficientes.
En general, le recomendamos que escriba funciones puras cada vez que sea posible y recurrir a las modificadoras solamente si hay una ventaja en usar este enfoque. Esto se denomina un **estilo de programación funcional**.

14.5. Desarrollo con prototipos vs. planificación

En este capítulo mostramos un enfoque de desarrollo de programas que denominamos **desarrollo con prototipos**. Para cada problema escribimos un bosquejo (o prototipo) que ejecutará el cálculo básico y lo probará en unos cuantos casos de prueba, corrigiendo errores a medida que surgen.
Aunque este enfoque puede ser efectivo, puede conducirnos a código innecesariamente complicado —ya que considera muchos casos especiales—y poco confiable—ya que es difícil asegurar que hemos descubierto todos los errores.
Una alternativa es el **desarrollo planificado**, en el que la profundización en el dominio del problema puede darnos una comprensión profunda que facilita bastante la programación. En el caso anterior, comprendimos que un objeto `Hora` realmente es un número de tres dígitos en base 60! El componente `segundos` contiene las "unidades," el componente `minutos` la "columna de sesentas," y el componente `hora` contiene la "columna de tres mil seiscientos."
Cuando escribimos `sumarHoras` e `incrementar`, realmente estábamos haciendo una suma en base 60, razón por la cual teníamos que efectuar un acarreo de una columna a la siguiente.
Esta observación sugiere otro enfoque al problema—podemos convertir un objeto `Hora` en un número único y aprovecharnos del hecho de que el computador sabe realizar aritmética. La siguiente función convierte un objeto `Hora` en un entero:

```
def convertirASegundos(t):
  minutos = t.hora * 60 + t.minutos
  segundos = minutos * 60 + t.segundos
  return segundos
```

Ahora necesitamos una forma de convertir desde entero a un objeto `Hora`:

```
def crearHora(segundos):
  h = Hora()
  h.hora = segundos/3600
  segundos = segundos - h.hora * 3600
  h.minutos = segundos/60
  segundos = segundos - h.minutos * 60
  h.segundos = segundos
  return h
```

Usted debe pensar unos minutos para convencerse de que esta técnica sí convierte, de una base a otra, correctamente. Asumiendo que ya está convencido, se pueden usar las funciones anteriores para reescribir `sumarHoras`:

```
def sumarHoras(t1, t2):
  segundos = convertirASegundos(t1) + convertirASegundos(t2)
  return crearHora(segundos)
```

Esta versión es mucho más corta que la original, y es mucho más fácil de demostrar que es correcta (asumiendo, como de costumbre, que las funciones que llama son correctas).

> *Como ejercicio, reescriba* `incrementar` *usando convertirASegundos y crearHora.*

14.6. Generalización

Desde cierto punto de vista, convertir de base 60 a base 10 y viceversa es más difícil que calcular solamente con horas. La conversión de bases es más abstracta, mientras que nuestra intuición para manejar horas está más desarrollada.

Pero si tenemos la intuición de tratar las horas como números en base 60 y hacemos la inversión de escribir las funciones de conversión (`convertirASegundos` y `crearHora`), obtenemos un programa más corto, legible, depurable y confiable.

También facilita la adición de más características. Por ejemplo, piense en el problema de restar dos `Hora`s para averiguar el tiempo que transcurre entre ellas. La solución ingenua haría resta llevando préstamos. En cambio, usar las funciones de conversión sería mas fácil.

Irónicamente, algunas veces el hacer de un problema algo más difícil (o más general) lo hace más fácil (porque hay menos casos especiales y menos oportunidades para caer en errores).

14.7. Algoritmos

Cuando usted escribe una solución general para una clase de problemas, en vez de encontrar una solución específica a un solo problema, ha escrito un **algoritmo**. Mencionamos esta palabra antes, pero no la definimos cuidadosamente. No es fácil de definir, así que intentaremos dos enfoques.

Primero, considere algo que no es un algoritmo. Cuando usted aprendió a multiplicar dígitos, probablemente memorizó la tabla de multiplicación. De hecho, usted memorizó 100 soluciones específicas. Este tipo de conocimiento no es algorítmico. Pero si usted era "perezoso," probablemente aprendió a hacer trampa por medio de algunos trucos. Por ejemplo, para encontrar el producto entre n y 9, usted puede escribir $n - 1$ como el primer dígito y $10 - n$ como el segundo. Este truco es una solución general para multiplicar cualquier dígito por el 9. ¡Este es un algoritmo!

Similarmente, las técnicas que aprendió para hacer suma con acarreo (llevando valores para la columna hacia la izquierda), resta con préstamos, y división larga, todas son algoritmos. Una de las características de los algoritmos es que no requieren inteligencia para ejecutarse. Son procesos mecánicos en el que cada paso sigue al anterior de acuerdo con un conjunto de reglas sencillas.

En nuestra opinión, es vergonzoso que los seres humanos pasemos tanto tiempo en la escuela aprendiendo a ejecutar algoritmos que, literalmente, no requieren inteligencia.
Por otro lado, el proceso de diseñar algoritmos es interesante, intelectualmente desafiante y una parte central de lo que denominamos programación.
Algunas cosas que la gente hace naturalmente sin dificultad o pensamiento consciente, son las mas difíciles de expresar algorítmicamente. Entender el lenguaje natural es una de ellas. Todos lo hacemos, pero hasta ahora nadie ha sido capaz de explicar *como* lo hacemos, al menos no con un algoritmo.

14.8. Glosario

Función pura: función que no modifica ninguno de los objetos que recibe como parámetros. La mayoría de las funciones puras son fructíferas.

Modificadora: función que cambia uno o varios de los objetos que recibe como parámetros. La mayoría de los modificadoras no retornan nada.

Estilo de programación funcional estilo de diseño de programas en el que la mayoría de funciones son puras.

Desarrollo con prototipos: es la forma de desarrollar programas empezando con un prototipo que empieza a mejorarse y probarse gradualmente.

Desarrollo planeado: es la forma de desarrollar programas que implica un conocimiento de alto nivel sobre el problema y mas planeación que el desarrollo incremental o el desarrollo con prototipos.

Algoritmo: conjunto de instrucciones para resolver una clase de problemas por medio de un proceso mecánico, no inteligente.

Capítulo 15

Clases y métodos

15.1. Características de orientación a objetos

Python es un **lenguaje de programación orientado a objetos**, lo que quiere decir que proporciona características que soportan la **programación orientada a objetos**. No es fácil definir la programación orientada a objetos, pero ya hemos notado algunos de sus elementos clave:

- Los programas se construyen a partir de definiciones de objetos y definiciones de funciones; la mayoría de los cómputos se hacen con base en objetos.

- Cada definición de objetos corresponde a algún concepto o cosa del mundo real, y las funciones que operan sobre esos objetos corresponden a las maneras en que los conceptos o cosas reales interactúan.

Por ejemplo, la clase `Hora`, definida en el Capítulo 14, corresponde a la forma en que la gente registra las horas del día y las funciones que definimos corresponden a la clase de cosas que la gente hace con horas. Similarmente, las clases `Punto` y `Rectangulo` corresponden a los conocidos conceptos geométricos.

Hasta aquí, no hemos aprovechado las características que Python proporciona para soportar la programación orientada a objetos. De hecho, estas características no son necesarias. La mayoría sólo proporciona una sintaxis alternativa para cosas que ya hemos logrado; pero, en muchos casos, esta forma alternativa es más concisa y comunica de una manera mas precisa la estructura de los programas.

Por ejemplo, en el programa `Hora` no hay una conexión obvia entre la definición de clase y las definiciones de funciones. Después de examinarlo un poco, es evidente que todas las funciones toman como parámetro al menos un objeto `Hora`.

Esta observación es la motivación para los **métodos**. Ya hemos visto algunos métodos como `keys` y `values`, que llamamos sobre diccionarios. Cada método se asocia con una clase y está pensado para invocarse sobre instancias de dicha clase.
Los métodos son como las funciones, pero con dos diferencias:

- Los métodos se definen adentro de una definición de clase, a fin de marcar explícitamente la relación entre la clase y éstos.
- La sintaxis para llamar o invocar un método es distinta que para las funciones.

En las siguientes secciones tomaremos las funciones de los capítulos anteriores y las transformaremos en métodos. Esta transformación es totalmente mecánica; se puede llevar a cabo siguiendo una secuencia de pasos. Si usted se siente cómodo al transformar de una forma a la otra, será capaz de escoger lo mejor de cada lado para resolver los problemas que tenga a la mano.

15.2. `imprimirHora`

En el capítulo 14, definimos una clase denominada `Hora` y usted escribió una función denominada `imprimirHora`, que lucía así:

```
class Hora:
  pass

def imprimirHora(h):
  print str(h.hora) + ":" +
        str(h.minutos) + ":" +
        str(h.segundos)
```

Para llamar esta función, le pasamos un objeto `Hora` como parámetro:

```
>>> horaActual = Hora()
>>> horaActual.hora = 9
>>> horaActual.minutos = 14
>>> horaActual.segundos = 30
>>> imprimirHora(horaActual)
```

Para convertir `imprimirHora` en un método todo lo que tenemos que hacer es ponerla adentro de la definición de clase. Note como ha cambiado la indentación.

```
class Hora:
  def imprimirHora(h):
```

```
    print str(h.hora) + ":" +
          str(h.minutos) + ":" +
          str(h.segundos)
```

Ahora podemos llamar a `imprimirHora` usando la notación punto.

```
>>> horaActual.imprimirHora()
```

Como de costumbre, el objeto en el que el método se llama aparece antes del punto y el nombre del método va a la derecha. El objeto al cual se invoca el método se asigna al primer parámetro, así que `horaActual` se asigna al parámetro `h`.
Por convención, el primer parámetro de un método se denomina `self` (en inglés, eso es algo como ”sí mismo”). La razón para hacerlo es un poco tortuosa, pero se basa en una metáfora muy útil.
La sintaxis para una llamada de función, `imprimirHora(horaActual)`, sugiere que la función es el agente activo. Dice algo como “Hey, `imprimirHora`! Aquí hay un objeto para que imprimas”.
En la programación orientada a objetos, los objetos son los agentes activos. Una invocación como `horaActual.imprimirHora()` dice algo como “Hey, objeto `horaActual`! Por favor, imprímase a sí mismo!”.
Este cambio de perspectiva parece ser sólo “cortesía”, pero puede ser útil. En los ejemplos que hemos visto no lo es. Pero, el transferir la responsabilidad desde las funciones hacia los objetos hace posible escribir funciones más versátiles y facilita la reutilización y el mantenimiento de código.

15.3. Otro ejemplo

Convirtamos `incrementar` (de la Sección 14.3) en un método. Para ahorrar espacio, omitiremos los métodos que ya definimos, pero usted debe conservarlos en su programa:

```
class Hora:
  # Las definiciones anteriores van aquí...

  def incrementar(self, segundos):
    hora.segundos = self.segundos + segundos

    if self.segundos >= 60:
      self.segundos = self.segundos - 60
      self.minutos = self.minutos + 1
```

```
    if self.minutos >= 60:
      self.minutos = self.minutos - 60
      self.hora = self.hora + 1

    return self
```

La transformación es totalmente mecánica —ponemos la definición del método adentro de la clase y cambiamos el nombre del primer parámetro.
Ahora podemos llamar a `incrementar` como método

```
horaActual.incrementar(500)
```

Nuevamente, el objeto con el cual se invoca el método se asigna al primer parámetro, `self`. El segundo parámetro, `segundos` recibe el valor `500`.

> *Como ejercicio, convierta `convertirASegundos` (de la Sección 14.5) a un método de la clase `Hora`.*

15.4. Un ejemplo más complejo

El método `despues` es un poco más complejo ya que opera sobre dos objetos `Hora`, no sólo uno. Solamente podemos convertir uno de los parámetros a `self`; el otro continúa igual:

```
class Hora:
  # Las definiciones anteriores van aqui...

  def despues(self, hora2):
    if self.hora > hora2.hora:
      return True
    if self.hora < hora2.hora:
      return False

    if self.minutos > hora2.minutos:
      return True
    if self.minutos < hora2.minutos:
      return False

    if self.segundos > hora2.segundos:
      return True
    return False
```

Llamamos a este método sobre un objeto y le pasamos el otro como argumento:

```
if horaComer.despues(horaActual):
  print "El pan estará listo para comer en un momento."
```

Casi se puede leer el llamado en lenguaje natural:"Si la hora para Comer viene despues de la hora Actual, entonces ...".

15.5. Argumentos opcionales

Hemos visto varias funciones primitivas que toman un número variable de argumentos. Por ejemplo, `string.find` puede tomar dos, tres o cuatro.
Es posible escribir funciones con listas de argumentos opcionales. Por ejemplo, podemos mejorar nuestra versión de `buscar` para que sea tan sofisticada como `string.find`.
Esta es la versión original que introdujimos en la Sección 8.7:

```
def buscar(cad, c):
  indice = 0
  while indice < len(cad):
    if cad[indice] == c:
      return indice
    indice = indice + 1
  return -1
```

Esta es la nueva versión, mejorada:

```
def buscar(cad, c,ini=0):
  indice = ini
  while indice < len(cad):
    if cad[indice] == c:
      return indice
    indice = indice + 1
  return -1
```

El tercer parámetro, `ini`, es opcional, ya que tiene un valor por defecto, `0`. Si llamamos a `buscar` con dos argumentos, se usa el valor por defecto y la búsqueda se hace desde el principio de la cadena:

```
>>> buscar("apple", "p")
1
```

Si se pasa el tercer parámetro, este **sobreescribe** el valor por defecto:

```
>>> buscar("apple", "p", 2)
2
>>> buscar("apple", "p", 3)
-1
```

> *Como ejercicio, añada un cuarto parámetro,* `fin`*, que especifique hasta donde continuar la búsqueda.*
>
> *Advertencia: Este ejercicio tiene una cascarita. El valor por defecto de* `fin` *debería ser* `len(cad)`*, pero esto no funciona. Los valores por defecto se evalúan en el momento de definición de las funciones, no cuando se llaman. Cuando se define* `buscar`*,* `cad` *no existe todavía, así que no se puede obtener su longitud.*

15.6. El método de inicialización

El **de inicialización** es un método especial que se llama cuando se crea un objeto. El nombre de este método es `__init__` (dos carácteres de subrayado, seguidos por `init`, y luego dos carácteres de subrayado más). Un método de inicialización para la clase `Hora` se presenta a continuación:

```
class Hora:
  def __init__(self, hora=0, minutos=0, segundos=0):
    self.hora = hora
    self.minutos = minutos
    self.segundos = segundos
```

No hay conflicto entre el atributo `self.hora` y el parámetro `hora`. La notación punto especifica a qué variable nos estamos refiriendo.
Cuando llamamos al método constructor de `Hora`, los argumentos se pasan a `init`:

```
>>> horaActual = Hora(9, 14, 30)
>>> horaActual.imprimirHora()
>>> 9:14:30
```

Como los parámetros son opcionales, se pueden omitir:

```
>>> horaActual = Hora()
>>> horaActual.imprimirHora()
>>> 0:0:0
```

O podemos pasar solo un parámetro:

```
>>> horaActual = Hora(9)
>>> horaActual.imprimirHora()
>>> 9:0:0
```

O, sólo los dos primeros:

```
>>> horaActual = Hora(9, 14)
>>> horaActual.imprimirHora()
>>> 9:14:0
```

Finalmente, podemos proporcionar algunos parámetros, nombrándolos explícitamente:

```
>>> horaActual = Hora(segundos = 30, hora = 9)
>>> horaActual.imprimirHora()
>>> 9:0:30
```

15.7. Reconsiderando la clase Punto

Reescribamos la clase `Punto` de la Sección 13.1 en un estilo más orientado a objetos:

```
class Punto:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

El método de inicialización toma los valores x y y como parámetros opcionales, el valor por defecto que tienen es 0.
El método `__str__`, retorna una representación de un objeto `Punto` en forma de cadena de texto. Si una clase proporciona un método denominado `__str__`, éste sobreescribe el comportamiento por defecto de la función primitiva `str`.

```
>>> p = Punto(3, 4)
>>> str(p)
'(3, 4)'
```

Imprimir un objeto `Punto` implícitamente invoca a `__str__` o sobre éste, así que definir a `__str__` también cambia el comportamiento de la sentencia `print`:

```
>>> p = Punto(3, 4)
>>> print p
(3, 4)
```

Cuando escribimos una nueva clase, casi siempre empezamos escribiendo `__init__`, ya que facilita la instanciación de objetos, y `__str__`, que casi siempre es esencial para la depuración.

15.8. Sobrecarga de operadores

Algunos lenguajes hacen posible cambiar la definición de los operadores primitivos cuando se aplican sobre tipos definidos por el programador. Esta característica se denomina **sobrecarga de operadores**. Es especialmente útil para definir tipos de datos matemáticos.
Por ejemplo, para sobrecargar el operador suma, `+`, proporcionamos un método denominado `__add__`:

```
class Punto:
  # los métodos definidos previamente van aquí...

  def __add__(self, otro):
    return Punto(self.x + otro.x, self.y + otro.y)
```

Como de costumbre, el primer parámetro es el objeto con el cual se invoca el método. El segundo parámetro se denomina con la palabra `otro` para marcar la distinción entre éste y `self`. Para sumar dos `Punto`s, creamos y retornamos un nuevo `Punto` que contiene la suma de las coordenadas en el eje x y la suma de las coordenadas en el eje y.
Ahora, cuando aplicamos el operador `+` a dos objetos `Punto`, Python hace el llamado del método `__add__`:

```
>>>   p1 = Punto(3, 4)
>>>   p2 = Punto(5, 7)
>>>   p3 = p1 + p2
>>>   print p3
(8, 11)
```

La expresión `p1 + p2` es equivalente a `p1.__add__(p2)`, pero luce mucho mas elegante.

> *Como ejercicio, agregue un método `__sub__(self, otro)` que sobrecargue el operador resta, y pruébelo*

Hay varias formas de sobrecargar el comportamiento del operador multiplicación: definiendo un método `__mul__`, o `__rmul__`, o ambos.

Si el operando izquierdo de `*` es un `Punto`, Python invoca a `__mul__`, asumiendo que el otro operando también es un `Punto`. El siguiente método calcula el **producto escalar** de los dos puntos de acuerdo a las reglas del álgebra lineal:

```
def __mul__(self, otro):
  return self.x * otro.x + self.y * otro.y
```

Si el operando izquierdo de `*` es un tipo primitivo y el operando derecho es un `Punto`, Python llama a `__rmul__`, que ejecuta la **multiplicación escalar** :

```
def __rmul__(self, otro):
  return Punto(otro * self.x,  otro * self.y)
```

El resultado ahora es un nuevo `Punto` cuyas coordenadas son múltiplos de las originales. Si `otro` pertenece a un tipo que no se puede multiplicar por un número de punto flotante, la función `__rmul__` producirá un error.

Este ejemplo ilustra las dos clases de multiplicación:

```
>>> p1 = Punto(3, 4)
>>> p2 = Punto(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

¿Que pasa si tratamos de evaluar `p2 * 2`? Ya que el primer parámetro es un `Punto`, Python llama a `__mul__` con `2` como segundo argumento. Dentro de `__mul__`, el programa intenta acceder al valor `x` de `otro`, lo que falla porque un número entero no tiene atributos:

```
>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
```

Desafortunadamente, el mensaje de error es un poco opaco. Este ejemplo demuestra una de las dificultades de la sobrecarga de operadores. Algunas veces es difícil saber qué código está ejecutándose.Para un ejemplo completo de sobrecarga de operadores vea el Apéndice B.

15.9. Polimorfismo

La mayoría de los métodos que hemos escrito sólo funcionan para un tipo de dato específico. Cuando se crea un nuevo tipo de objeto, se escriben métodos que operan sobre ese tipo.

Pero hay ciertas operaciones que se podrían aplicar a muchos tipos, un ejemplo de éstas son las operaciones aritméticas de las secciones anteriores. Si muchos tipos soportan el mismo conjunto de operaciones, usted puede escribir funciones que trabajen con cualquiera de estos tipos.

Por ejemplo la operación `multsuma` (que se usa en el álgebra lineal) toma tres parámetros, multiplica los primeros dos y luego suma a esto el tercero. En Python se puede escribir así:

```
def multsuma (x, y, z):
  return x * y + z
```

Este método funcionará para cualesquier valores de `x` e `y` que puedan multiplicarse, y para cualquier valor de `z` que pueda sumarse al producto.

Podemos llamarla sobre números:

```
>>> multsuma (3, 2, 1)
7
```

O sobre `Punto`s:

```
>>> p1 = Punto(3, 4)
>>> p2 = Punto(5, 7)
>>> print multsuma (2, p1, p2)
(11, 15)
>>> print multsuma (p1, p2, 1)
44
```

En el primer caso, el `Punto` se multiplica por un escalar y luego se suma a otro `Punto`. En el segundo caso, el producto punto produce un valor numérico, así que el tercer parámetro también tiene que ser un número.

Una función como ésta, que puede tomar parámetros con tipos distintos se denomina **polimórfica**.

Otro ejemplo es la función `derechoyAlReves`, que imprime una lista dos veces, al derecho y al revés:

```
def derechoyAlReves(l):
  import copy
  r = copy.copy(l)
  r.reverse()
  print str(l) + str(r)
```

Como el método `reverse` es una función modificadora, tenemos que tomar la precaución de hacer una copia de la lista antes de llamarlo. De esta forma la lista que llega como parámetro no se modifica.
Aquí hay un ejemplo que aplica `derechoyAlReves` a una lista:

```
>>>   miLista = [1, 2, 3, 4]
>>>   derechoyAlReves(miLista)
[1, 2, 3, 4][4, 3, 2, 1]
```

Por supuesto que funciona para listas, esto no es sorprendente. Lo que sería sorprendente es que pudiéramos aplicarla a un `Punto`.
Para determinar si una función puede aplicarse a un nuevo tipo de dato usamos la regla fundamental del polimorfismo:

> **Si todas las operaciones adentro de la función pueden aplicarse al otro tipo, la función puede aplicarse al tipo.**

Las operaciones que usa el método son `copy`, `reverse`, y `print`.
`copy` funciona para cualquier objeto, y como ya hemos escrito un método `__str__` para los `Punto`s, lo único que nos falta es el método `reverse` dentro de la clase `Punto`:

```
def reverse(self):
  self.x , self.y = self.y, self.x
```

Entonces podemos aplicar `derechoyAlReves` a objetos `Punto`:

```
>>>   p = Punto(3, 4)
>>>   derechoyAlReves(p)
(3, 4)(4, 3)
```

El mejor tipo de polimorfismo es el que no se pretendía lograr, aquel en el que se descubre que una función escrita puede aplicarse a un tipo, para el cual no se había planeado hacerlo.

15.10. Glosario

Lenguaje orientado a objetos: lenguaje que tiene características, como las clases definidas por el usuario y la herencia, que facilitan la programación orientada a objetos.

Programación orientada a objetos: estilo de programación en el que los datos y las operaciones que los manipulan se organizan en clases y métodos.

Método: función que se define adentro de una clase y se llama sobre instancias de ésta.

Sobreescribir: reemplazar un valor preexistente. Por ejemplo, se puede reemplazar un parámetro por defecto con un argumento particular y un método ya definido, proporcionando un nuevo método con el mismo nombre.

Método de inicialización: método especial que se llama automáticamente cuando se crea un nuevo objeto. Inicializa los atributos del objeto.

Sobrecarga de operadores: extender el significado de los operadores primitivos (`+`, `-`, `*`, `>`, `<`, etc.) de forma que acepten tipos definidos por el usuario.

Producto punto: operación del álgebra lineal que multiplica dos `Punto`s y produce un valor numérico.

Multiplicación escalar: operación del álgebra lineal que multiplica cada una de las coordenadas de un `Punto` por un valor numérico.

Polimórfica: función que puede operar sobre varios tipos de datos. Si todas las operaciones que se llaman dentro de la función se le pueden aplicar al tipo de dato, entonces la función puede aplicársela al tipo.

Capítulo 16

Conjuntos de objetos

16.1. Composición

En este momento usted ya ha visto varios ejemplos de composición. Uno de los primeros fue una invocación de un método como parte de una expresión. Otro ejemplo es la estructura anidada de sentencias; por ejemplo, se puede colocar una sentencia `if` dentro de un ciclo `while`, dentro de otra sentencia `if`.

Después de observar esto y haber aprendido sobre listas y objetos no debería sorprenderse al saber que se pueden crear listas de objetos. También pueden crearse objetos que contengan listas (como atributos), listas que contengan listas, objetos que contengan objetos, y así sucesivamente.

En este capítulo y el siguiente, mostraremos algunos ejemplos de estas combinaciones, usando objetos `Carta`.

16.2. Objeto `Carta`

Si usted no tiene familiaridad con juegos de cartas este es un buen momento para conseguir una baraja, de lo contrario este capítulo no tendrá mucho sentido. Hay cincuenta y dos cartas en una baraja, cada una pertenece a una de las cuatro figuras y uno de los trece valores. Las figuras son Picas, Corazones, Diamantes y Tréboles. Los valores son As, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K. Dependiendo del juego, el valor del As puede ser más alto que el de un rey o más bajo que 2.

Si deseamos definir un nuevo objeto para representar una carta del naipe, parece obvio que los atributos deberían ser `valor` y `figura`. No es tan obvio que tipo de dato asignar a estos atributos. Una posibilidad consiste en usar cadenas de texto con palabras como `Picas` para las figuras y `Reina` para los valores. Un problema

de esta implementación es que no sería tan fácil comparar cartas para ver cuál tiene un valor mayor o una figura mayor.

Una alternativa consiste en usar enteros para **codificar** los valores y las figuras. Por "codificar", no estamos haciendo alusión a encriptar o traducir a un código secreto. Lo que un científico de la computación considera "codificar" es "definir una correspondencia entre una secuencia de números y los objetos que deseamos representar". Por ejemplo:

Picas	$\mapsto$	3
Corazones	$\mapsto$	2
Diamantes	$\mapsto$	1
Tréboles	$\mapsto$	0

Una característica notable de esta correspondencia es que las figuras aparecen en orden decreciente de valor así como los enteros van disminuyendo. De esta forma, podemos comparar figuras mediante los enteros que las representan. Una correspondencia para los valores es bastante sencilla; cada número se relaciona con el entero correspondiente, y para las cartas que se representan con letras tenemos lo siguiente:

A	$\mapsto$	1
J	$\mapsto$	11
Q	$\mapsto$	12
K	$\mapsto$	13

La razón para usar notación matemática en estas correspondencias es que ellas no hacen parte del programa en Python. Son parte del diseño, pero nunca aparecen explícitamente en el código fuente. La definición de la clase `Carta` luce así:

```
class Carta:
  def __init__(self, figura=0, valor=0):
    self.figura = figura
    self.valor = valor
```

Como de costumbre, proporcionamos un método de inicialización que toma un parámetro opcional para cada atributo.

Para crear un objeto que represente el 3 de tréboles, usamos este comando:

```
tresTreboles = Carta(0, 3)
```

El primer argumento, `0`, representa la figura (tréboles).

16.3. Atributos de clase y el método `__str__`

Para imprimir objetos `Carta` en una forma que la gente pueda leer fácilmente, queremos establecer una correspondencia entre códigos enteros y palabras. Una

forma natural de hacerlo es con listas de cadenas de texto. Asignamos estas listas a **atributos de clase** al principio de la clase:

```
class Carta:
  listaFiguras = ["Treboles", "Diamantes", "Corazones",
  "Picas"]
  listaValores = ["narf", "As", "2", "3", "4", "5", "6",
  "7","8", "9", "10", "Jota", "Reina", "Rey"]

  # se omite el metodo init

  def __str__(self):
    return (self.listaFiguras[self.valor] + " de " +
            self.listaValores[self.figura])
```

Un atributo de clase se define afuera de los métodos y puede ser accedido desde cualquiera de ellos.
Dentro de `__str__`, podemos usar a `listaFiguras` y `listaValores` para establecer una correspondencia entre los valores numéricos de `figura`, `valor` y los nombres de las cartas. La expresión `self.listaFiguras[self.figura]` significa "use el atributo `figura` del objeto `self` como índice dentro del atributo de clase `listaFiguras`, esto seleccionará la cadena de texto apropiada".
La razón para el `"narf"` en el primer elemento de `listaValores` consiste en ocupar el elemento cero de la lista que no va a ser usado en el programa. Los valores válidos van de 1 a 13. Este elemento desperdiciado no es necesario, podríamos haber empezado a contar desde 0, pero es menos confuso codificar 2 como 2, 3 como 3 ... y 13 como 13.
Con los métodos que tenemos hasta aquí, podemos crear e imprimir cartas:

```
>>> c1 = Carta(1, 11)
>>> print c1
Jota de Diamantes
```

Los atributos de clase como `listaFiguras` se comparten por todos los objetos `Carta`. La ventaja de esto es que podemos usar cualquier objeto `Carta` para acceder a ellos:

```
>>> c2 = Carta(1, 3)
>>> print c2
3 de Diamantes
>>> print c2.listaFiguras[1]
Diamantes
```

La desventaja es que si modificamos un atributo de clase, afecta a todas las otras instancias de la clase. Por ejemplo, si decidimos que “Jota de Diamantes” debería llamarse “Caballero de Rombos rojos,” podríamos ejecutar:

```
>>> c1.listaFiguras[1] = "Caballero de Rombos rojos"
>>> print c1
Caballero de Rombos rojos
```

El problema es que *todos* los Diamantes ahora son Rombos rojos:

```
>>> print c2
3 de Rombos rojos
```

Usualmente no es una buena idea modificar los atributos de clase.

16.4. Comparando cartas

Para los tipos primitivos contamos con los operadores (<, >, ==, etc.) que determinan cuándo un valor es mayor, menor, mayor o igual, menor o igual, o igual al otro. Para los tipos definidos por el programador podemos sobrecargar el comportamiento de los operadores predefinidos proporcionando un método llamado `__cmp__`. Por convención, `__cmp__` toma dos parámetros, `self` y `otro`, y retorna 1 si el primer objeto es más grande, -1 si el segundo es más grande y 0 si son iguales entre si.

Algunos tipos tienen un orden total, lo que quiere decir que cualquier pareja de elementos se puede comparar para decidir cuál de ellos es mayor. Por ejemplo, los números enteros y los de punto flotante tienen un orden total. Algunos conjuntos no tienen relación de orden, lo que quiere decir que no hay una manera sensata de determinar que un elemento es mayor que otro. Por ejemplo, las frutas no tienen una relación de orden, y esta es la razón por la que no se pueden comparar manzanas con naranjas.

El conjunto de cartas tiene un orden parcial, lo que quiere decir que algunas veces se pueden comparar elementos, y otras veces no. Por ejemplo, el 3 de Picas es mayor que el 2 de picas, y el 3 de Diamantes es mayor que el 3 de Picas. Pero, ¿que es más alto, el 3 de Picas o el 2 de Diamantes? Uno tiene un valor más alto, pero el otro tiene una figura más alta.

Para lograr comparar las cartas, hay que tomar una decisión sobre la importancia del valor y de la figura. Para ser honestos, esta decisión es arbitraria. Así que tomaremos la opción de determinar qué figura es más importante, basándonos en que un mazo de cartas nuevo viene con las Picas (en orden), luego los Diamantes, y así sucesivamente.

Con esta decisión `__cmp__` queda así:

```
def __cmp__(self, otro):
  # chequea las figuras
  if self.figura > otro.figura: return 1
  if self.figura < otro.figura: return -1
  # Si tienen la misma figura...
  if self.valor > otro.valor: return 1
  if self.valor < otro.valor: return -1
  # si los valores son iguales... hay un empate
  return 0
```

Con este orden los Ases valen menos que los Dos.

> *Como ejercicio, modifique* `__cmp__` *para que los Ases tengan mayor puntaje que los reyes.*

16.5. Mazos

Ahora que tenemos objetos para representar `Carta`s, el siguiente paso lógico consiste en definir una clase para representar un `Mazo`. Por supuesto, un mazo (o baraja) está compuesto por cartas, así que cada instancia de `Mazo` contendrá como atributo una lista de cartas.
La siguiente es la definición de la clase `Mazo`. El método de inicialización crea el atributo `cartas` y genera el conjunto usual de cincuenta y dos cartas:

```
class Mazo:
  def __init__(self):
    self.cartas = []
    for figura in range(4):
      for valor in range(1, 14):
        self.cartas.append(Carta(figura, valor))
```

La forma más sencilla de llenar el mazo consiste en usar un ciclo anidado. El ciclo exterior enumera las figuras de 0 a 3. El ciclo interno enumera los valores de 1 a 13. Como el ciclo exterior itera cuatro veces y el interno itera trece veces, el número total de iteraciones es cincuenta y dos (4×13). Cada iteración crea una nueva instancia de `Carta` y la pega a la lista `cartas`.
El método `append` acepta secuencias mutables como las listas y no acepta tuplas.

16.6. Imprimiendo el mazo

Como de costumbre, cuando definimos un nuevo tipo de objeto, deseamos tener un método que imprima su contenido. Para imprimir un `Mazo`, recorremos la lista e imprimimos cada objeto `Carta`:

```
class Mazo:
  ...
  def imprimirMazo(self):
    for carta in self.cartas:
      print carta
```

En este ejemplo y en los que siguen, los puntos suspensivos indican que hemos omitido los otros métodos de la clase.
Otra alternativa a `imprimirMazo` puede ser escribir un método `__str__` para la clase `Mazo`. La ventaja de `__str__` radica en su mayor flexibilidad. Además de imprimir el contenido del objeto, genera una representación de él en una cadena de texto que puede manipularse en otros lugares del programa, incluso antes de imprimirse.
A continuación hay una versión de `__str__` que retorna una representación de un `Mazo`. Para añadir un estilo de cascada, cada carta se imprime un espacio mas hacia la derecha que la anterior:

```
class Mazo:
  ...
  def __str__(self):
    s = ""
    for i in range(len(self.cartas)):
      s = s + " "*i + str(self.cartas[i]) + "\n"
    return s
```

Este ejemplo demuestra varios puntos. Primero, en vez de recorrer los elementos de la lista `self.cartas`, estamos usando a `i` como variable de ciclo que lleva la posición de cada elemento en la lista de cartas.
Segundo, estamos usando el operador multiplicación aplicado a un número y una cadena, de forma que la expresión `" "*i` produce un número de espacios igual al valor actual de `i`.
Tercero, en vez de usar el comando `print` para realizar la impresión, utilizamos la función `str`. Pasar un objeto como argumento a `str` es equivalente a invocar el método `__str__` sobre el objeto.
Finalmente, estamos usando a la variable `s` como **acumulador**. Inicialmente `s` es la cadena vacía. En cada iteración del ciclo se genera una nueva cadena y se concatena con el valor viejo de `s` para obtener el nuevo valor. Cuando el ciclo finaliza, `s` contiene la representación completa del `Mazo`, que se despliega (parcialmente) así:

```
>>> mazo = Mazo()
>>> print mazo
```

```
As de Picas
 2 de Picas
  3 de Picas
   4 de Picas
    5 de Picas
     6 de Picas
      7 de Picas
       8 de Picas
        9 de Picas
         10 de Picas
          J de Picas
           Reina de Picas
            Rey de Picas
             As de Diamantes
```

Aunque el resultado se despliega en 52 líneas, es una sola cadena que contiene carácteres nueva linea (`\n`).

16.7. Barajando el mazo

Si un mazo se baraja completamente, cualquier carta tiene la misma probabilidad de aparecer en cualquier posición, y cualquier posición tiene la misma probabilidad de contener a cualquier carta.
Para barajar el mazo, usamos la función `randrange` que pertenece al módulo del sistema `random`. `randrange` recibe dos parámetros enteros `a` y `b`, y se encarga de escoger al azar un valor perteneciente al rango `a <= x <b`. Como el límite superior es estrictamente menor que `b`, podemos usar el número de elementos de una lista como el segundo parámetro y siempre obtendremos un índice válido como resultado. Por ejemplo, esta expresión escoge al azar el índice de una carta en un mazo:

```
random.randrange(0, len(self.cartas))
```

Una manera sencilla de barajar el mazo consiste en recorrer todas las cartas intercambiando cada una con otra carta escogida al azar. Es posible que la carta se intercambie consigo misma, pero esto no causa ningún problema. De hecho, si prohibiéramos esto, el orden de las cartas no sería tan aleatorio:

```
class Mazo:
  ...
```

```
  def barajar(self):
    import random
    nCartas = len(self.cartas)
    for i in range(nCartas):
      j = random.randrange(i, nCartas)
      self.cartas[i], self.cartas[j] = self.cartas[j],
                                       self.cartas[i]
```

En vez de asumir que hay 52 cartas en el mazo, obtenemos el número de ellas a través de la función len y lo almacenamos en la variable `nCartas`.
Para cada carta en el mazo, escogemos, aleatoriamente, una carta de las que no han sido barajadas todavía. Intercambiamos la carta actual (con índice `i`) con la seleccionada (con índice `j`). Para intercambiar las cartas usamos asignación de tuplas, como en la sección 10.2:

```
self.cartas[i], self.cartas[j] = self.cartas[j],
                                 self.cartas[i]
```

Como ejercicio, reescriba este intercambio sin usar asignación de tuplas.

16.8. Eliminando y entregando cartas

Otro método que sería útil para la clase `Mazo` es `eliminarCarta`, que toma una carta como parámetro, la remueve y retorna True si la encontró en el mazo o False si no estaba:

```
class Mazo:
  ...
  def eliminarCarta(self, carta):
    if carta in self.cartas:
      self.cartas.remove(carta)
      return True
    else:
      return True
```

El operador `in` retorna True si el primer operando se encuentra dentro del segundo, que debe ser una secuencia. Si el primer operando es un objeto, Python usa el método `__cmp__` para determinar la igualdad de elementos en la lista. Como la función `__cmp__` en la clase `Carta` detecta igualdad profunda, el método `eliminarCarta` detecta igualdad profunda.
Para entregar cartas necesitamos eliminar y retornar la primera carta del mazo. El método `pop` de las listas proporciona esta funcionalidad:

```
class Mazo:
  ...
  def entregarCarta(self):
    return self.cards.pop()
```

En realidad, `pop` elimina la *última* carta de la lista, así que realmente estamos entreganado cartas por la parte inferior, y esto no causa ningún inconveniente.
Una operación más que podemos requerir es la función booleana `estaVacio`, que retorna True si el mazo está vacío:

```
class Mazo:
  ...
  def estaVacio(self):
    return (len(self.cartas) == 0)
```

16.9. Glosario

Codificar: representar un conjunto de valores usando otro conjunto de valores estableciendo una correspondencia entre ellos.

Atributo de clase: variable de una clase que está fuera de todos los métodos. Puede ser accedida desde todos los métodos y se comparte por todas las instancias de la clase.

Acumulador: variable que se usa para acumular una serie de valores en un ciclo. Por ejemplo, concatenar varios valores en una cadena o sumarlos.

Capítulo 17

Herencia

17.1. Definición

La característica más asociada con la programación orientada a objetos quizás sea la **herencia**. Ésta es la capacidad de definir una nueva clase que constituye la versión modificada de una clase preexistente.

La principal ventaja de la herencia consiste en que se pueden agregar nuevos métodos a una clase sin modificarla. El nombre "herencia" se usa porque la nueva clase hereda todos los métodos de la clase existente. Extendiendo esta metáfora, la clase preexistente se denomina la clase **madre**. La nueva clase puede llamarse clase **hija** o, "subclase".

La herencia es muy poderosa. Algunos programas complicados se pueden escribir de una manera más sencilla y compacta a través del uso de la herencia. Además, facilita la reutilización de código, ya que se puede especializar el comportamiento de una clase madre sin modificarla. En algunos casos, las relaciones entre las clases reflejan la estructura de las entidades del mundo real que se presentan en un problema, y esto hace que los programas sean más comprensibles.

Por otra parte, la herencia puede dificultar la lectura de los programas. Cuando un método se llama, puede que no sea muy claro donde está definido. El código relevante puede estar disperso entre varios módulos. Además, muchas soluciones que se pueden escribir con el uso de herencia, también se pueden escribir sin ella, y de una manera elegante (algunas veces más elegante). Si la estructura natural de las entidades que intervienen en un problema no se presta a pensar en términos de herencia, este estilo de programación puede causar más perjuicios que beneficios.

En este capítulo demostraremos el uso de la herencia como parte de un programa que implementa el juego de cartas La Solterona. Una de las metas es escribir una base de código que pueda reutilizarse para implementar otros juegos de cartas.

17.2. Una mano de cartas

Para casi todos los juegos de cartas vamos a requerir representar una mano de cartas. Hay una clara semejanza entre un mazo y una mano de cartas, ambos son conjuntos de cartas y requieren operaciones como agregar o eliminar cartas. También, podríamos barajar mazos y manos de cartas.

Por otro lado, una mano se diferencia de un mazo; dependiendo del juego que estemos considerando podríamos ejecutar algunas operaciones sobre las manos que no tienen sentido en un mazo. Por ejemplo, en el póker, podríamos clasificar manos como un par, una terna, 2 pares, escalera, o podríamos comparar un par de manos. En el bridge, podríamos calcular el puntaje para una mano con el objetivo de hacer una apuesta.

Esta situación sugiere el uso de la herencia. Si una `Mano` es una subclase de `Mazo`, tendrá todos los métodos definidos en ésta y se podrán agregar nuevos métodos.

En la definición de una clase hija, el nombre de la clase madre se encierra entre paréntesis:

```
class Mano(Mazo):
  pass
```

Esta sentencia indica que la nueva clase `Mano` hereda de la clase `Mazo`.

El constructor de `Mano` inicializa los atributos `nombre` y `cartas`. La cadena `nombre` identifica cada mano, probablemente con el nombre del jugador que la sostiene. Por defecto, el parámetro nombre es una cadena vacía. `cartas` es la lista de cartas en la mano, inicializada a una lista vacía:

```
class Mano(Mazo):
  def __init__(self, nombre=""):
    self.cartas = []
    self.nombre = nombre
```

Para la gran mayoría de juegos de cartas es necesario agregar y remover cartas del mazo. Como `Mano` hereda `eliminarCarta` de `Mazo`, ya tenemos la mitad del trabajo hecho, solo tenemos que escribir `agregarCarta`:

```
class Mano(Mazo):
  ...
  def agregarCarta(self,carta) :
    self.cartas.append(carta)
```

Recuerde que la elipsis (...) indica que estamos omitiendo los otros métodos. El método `append` de las listas permite agregar la nueva carta.

17.3. Repartiendo cartas

Ahora que tenemos una clase `Mano` deseamos transferir cartas del `Mazo` a las manos. No es fácil decidir que el método que implemente esta transferencia se incluya en la clase `Mano` o en la clase `Mazo`, pero como opera sobre un solo mazo y (probablemente) sobre varias manos, es más natural incluirlo en `Mazo`.

El método `repartir` debería ser muy general, ya que diferentes juegos tienen diversas reglas. Podríamos transferir todo el mazo de una vez, o repartir carta a carta alternando por cada mano, como se acostumbra en los casinos.

El método `repartir` toma dos parámetros, una lista (o tupla) de manos y el número total de cartas que se deben entregar. Si no hay suficientes cartas en el mazo para repartir, entrega todas las cartas y se detiene:

```
class Mazo:
  ...
  def repartir(self, manos, nCartas=999):
    nManos = len(manos)
    for i in range(nCartas):
      if self.estaVacio():
            break    # se detiene si no hay cartas
      # toma la carta en el tope
      carta = self.eliminarCarta()
      # siguiente turno!
      mano = manos[i % nManos]
      # agrega la carta a la mano
      mano.agregarCarta(carta)
```

El segundo parámetro, `nCartas`, es opcional; y por defecto es un número entero grande que garantice que todas las cartas del mazo se repartan.

La variable de ciclo `i` va de 0 a `nCartas-1`. En cada iteración remueve una carta al tope del mazo usando el método `pop`, que elimina y retorna el último elemento de la lista.

El operador residuo (`%`) nos permite repartir cartas de manera circular (una carta a una mano distinta en cada iteración). Cuando `i` es igual al número de manos en la lista, la expresión `i% nManos` *da la vuelta* retornando la primera posición de la lista (la posición 0).

17.4. Imprimiendo una mano

Para imprimir el contenido de una mano podemos aprovechar los métodos `__str__` e `imprimirMazo` heredados de `Mazo`. Por ejemplo:

```
>>> mazo = Mazo()
>>> mazo.barajar()
>>> mano = Mano("Rafael")
>>> mazo.repartir([mano], 5)
>>> print mano
La Mano Rafael contiene
2 de Picas
 3 de Picas
  4 de Picas
   As de Corazones
    9 de tréboles
```

No es una gran mano, pero se puede mejorar.
Aunque es conveniente heredar los métodos existentes, hay un dato adicional que un objeto `Mano` puede incluir cuando se imprime, para lograr esto implementamos `__str__` sobrecargando el que está definido en la clase `Mazo`:

```
class Mano(Mazo)
  ...
  def __str__(self):
    s = "Mano " + self.nombre
    if self.estaVacio():
      s = s + " esta vacia\n"
    else:
      s = s + " contiene\n"
    return s + Mazo.__str__(self)
```

Inicialmente, `s` es una cadena que identifica la mano. Si está vacía, se añade la cadena `esta vacia`. Si esto no es así se añade la cadena `contiene` y la representación textual de la clase `Mazo`, que se obtiene aplicando el método `__str__` a `self`.
Parece extraño aplicar el método `__str__` de la clase `Mazo` a `self` que se refiere a la `Mano` actual. Para disipar cualquier duda, recuerde que `Mano` es una clase de `Mazo`. Los objetos `Mano` pueden hacer todo lo que los objetos `Mazo` hacen, así que esto es legal.
En general, siempre se puede usar una instancia de una subclase en vez de una instancia de la clase padre.

17.5. La clase JuegoDeCartas

La clase JuegoDeCartas se encarga de algunas operaciones básicas comunes a todos los juegos, tales como crear el mazo y barajarlo:

```
class JuegoDeCartas:
  def __init__(self):
    self.mazo = Mazo()
    self.mazo.barajar()
```

En este ejemplo vemos que la inicialización realiza algo más importante que asignar valores iniciales a los atributos.

Para implementar un juego específico podemos heredar de JuegoDeCartas y agregar las características del juego particular que estemos desarrollando. A manera de ejemplo, escribiremos una simulación del juego La Solterona.

El objetivo de La Solterona es deshacerse de todas las cartas. Cada jugador hace esto emparejando cartas por figura y valor. Por ejemplo el 4 de Treboles se empareja con el 4 de Picas, por que son cartas negras. La J de Corazones se empareja con la J de Diamantes, porque son cartas rojas.

Para empezar, la reina de Treboles se elimina del mazo, de forma que la reina de Picas no tenga pareja. Las otras 51 cartas se reparten equitativamente entre los jugadores. Después de repartir cada jugador busca parejas en su mano y las descarta. Cuando ningún jugador pueda descartar más se empieza a jugar por turnos. Cada jugador escoge una carta de su vecino a la izquierda (sin mirarla). Si la carta escogida se empareja con alguna carta en la mano del jugador, el par se elimina. Si esto no es así, la carta debe agregarse a la mano del jugador que escoge. Poco a poco, se realizarán todos los emparejamientos posibles, dejando únicamente a la reina de Picas en la mano del perdedor.

En nuestra simulación del juego, el computador juega todas las manos. Desafortunadamente, algunos matices del juego real se pierden en la simulación por computador. En un juego, el jugador con la Solterona intenta deshacerse de ella de diferentes formas, ya sea desplegándola de una manera prominente, o ocultándola de alguna manera. El programa simplemente escogerá una carta de algún vecino aleatoriamente.

17.6. La clase ManoSolterona

Una mano para jugar a la Solterona requiere algunas capacidades que no están presentes en la clase Mano. Vamos a definir una nueva clase ManoSolterona, que hereda de Mano y provee un método adicional llamado quitarPareja:

```
class ManoSolterona(Mano):
```

```
  def quitarPareja(self):
    conteo = 0
    cartasOriginales = self.cartas[:]
    for carta in cartasOriginales:
      m = Carta(3 - carta.figura, carta.valor)
      if pareja in self.cartas:
        self.cartas.remove(carta)
        self.cartas.remove(m)
        print "Mano %s: %s se empareja con %s" %
              (self.name,carta,m)
        cont = cont + 1
    return cont
```

Empezamos haciendo una copia de la lista de cartas, de forma que podamos recorrerla y simultaneamente eliminar cartas. Como `self.cartas` se modifica en el ciclo, no vamos a utilizarlo para controlar el recorrido. ¡Python puede confundirse totalmente si empieza a recorrer una lista que está cambiando!

Para cada carta en la mano, averiguamos si se empareja con una carta escogida de la mano de otra persona. Para esto, tienen que tener el mismo valor y la otra figura del mismo color. La expresión `3 - carta.figura` convierte un trébol (figura 0) en una Pica (figura 3) y a un Diamante (figura 1) en un Corazón (figura 2). Usted puede comprobar que las operaciones inversas también funcionan. Si hay una carta que se empareje, las dos se eliminan.

El siguiente ejemplo demuestra cómo usar `quitarPareja`:

```
>>> juego = JuegoDeCartas()
>>> mano = ManoSolterona("frank")
>>> juego.mazo.repartir([mano], 13)
>>> print mano
Mano frank contiene
As de Picas
 2 de Diamantes
  7 de Picas
   8 de Treboles
    6 de Corazones
     8 de Picas
      7 de Treboles
       Reina de Treboles
        7 de Diamantes
         5 de Treboles
          Jota de Diamantes
           10 de Diamantes
            10 de Corazones
```

```
>>> mano.quitarPareja()
Mano frank: 7 de Picas se empareja con 7 de Treboles
Mano frank: 8 de Picas se empareja con 8 de Treboles
Mano frank: 10 de Diamantes se empareja con 10 de Corazones
>>> print mano
Mano frank contiene
Ace de Picas
 2 de Diamantes
  6 de Corazones
   Reina de Treboles
    7 de Diamantes
     5 de Treboles
      Jota de Diamantes
```

Tenga en cuenta que no hay método `__init__` en la clase `ManoSolterna`. Lo heredamos de `Mano`.

17.7. La clase JuegoSolterona

Ahora podemos dedicarnos a desarrollar el juego. `JuegoSolterona` es una subclase de `JuegoDeCartas` con un nuevo método llamado `jugar` que recibe una lista de jugadores como parámetro.
Como `__init__` se hereda de `JuegoDeCartas`, tendremos la garantía de que un objeto de tipo `JuegoSolterona` contendrá un mazo barajado:

```
class JuegoSolterona(JuegoDeCartas):
  def jugar(self, nombres):
    # elimina la Reina de Treboles
    self.mazo.eliminarCarta(Carta(0,12))

    # prepara una mano para cada jugador
    self.manos = []
    for nombre in nombres :
      self.manos.append(ManoJuegoSolterona(nombre))

    # reparte las  cartas
    self.mazo.repartir(self.cartas)
    print "---------- Cartas repartidas!"
    self.imprimirManos()

    # quitar parejas iniciales
```

```
    parejas = self.eliminarParejas()
    print "---------- Parejas descartadas, comienza el juego"
    self.imprimirManos()

    # jugar hasta que las 50 cartas sean descartadas
    turno = 0
    numManos = len(self.manos)
    while parejas < 25:
      parejas = parejas+self.jugarUnTurno(turno)
      turno = (turno + 1) % numManos

    print "-------- Juego terminado"
    self.printManos()
```

Algunos de las etapas del juego se han separado en métodos. `eliminarParejas` recorre la lista de manos invocando `eliminarPareja` en cada una de ellas:

```
class JuegoSolterona(JuegoDeCartas):
  ...
  def eliminarParejas(self):
    count = 0
    for mano in self.manos:
      cont = cont + mano.eliminarParejas()
    return count
```

Como ejercicio, escriba `imprimaManos` *que recorre la lista* `self.manos` *e imprime cada mano.*

`cont` es un acumulador que lleva cuenta del número de parejas que se encontraron en cada mano.
Cuando el número total de parejas encontradas llega a ser veinticinco, se han quitado cincuenta cartas de las manos, y esto implica que la única carta que resta es la reina de Picas y que el juego ha terminado.
La variable `turno` lleva la pista de cual es el jugador que tiene el turno para jugar. Empieza en 0 y se incrementa de uno en uno; cuando alcanza el valor `numManos`, el operador residuo lo reinicia en 0.
El método `jugarUnTurno` toma un parámetro que indica el jugador que tiene el turno. El valor de retorno es el número de parejas encontradas durante este turno:

```
class JuegoSolterona(JuegoDeCartas):
```

```
  ...
  def jugarUnTurno(self, i):
    if self.manos[i].estaVacia():
      return 0
    vecino = self.encontrarVecino(i)
    cartaEscogida = self.manos[vecino].eliminarCarta()
    self.manos[i].agregarCarta(cartaEscogida)
    print "Mano", self.manos[i].nombre,
          "escogió", cartaEscogida
    cont = self.manos[i].eliminarParejas()
    self.manos[i].barajar()
    return cont
```

Si la mano de un jugador está vacía, este jugador está fuera del juego, así que no hace ninguna acción y retorna 0.
Si esto no es así, un turno consiste en encontrar el primer jugador en la izquierda que tenga cartas, tomar una de él, y buscar por parejas. Antes de retornar, las cartas en la mano se barajan para que la elección del siguiente jugador sea al azar.
El método `encontrarVecino` comienza con el jugador a la izquierda y continua buscando de manera circular hasta que encuentra un jugador que tenga cartas:

```
class JuegoSolterona(JuegoDeCartas):
  ...
  def encontrarVecino(self, i):
    numManos = len(self.manos)
    for siguiente in range(1,numManos):
      vecino = (i + siguiente) % numManos
      if not self.manos[vecino].estaVacia():
        return vecino
```

Si `encontrarVecino` diera toda la vuelta sin encontrar cartas, retornaría `None` y causaría un error en algún otro lugar del programa. Afortunadamente, usted puede comprobar que esto nunca va a pasar (siempre y cuando el fin del juego se detecte correctamente).
Hemos omitido el método `imprimaManos`. Usted puede escribirlo.
La siguiente salida es de un juego en el que solo las primeras 15 cartas mas altas (con valor 10 y superior) se repartieron a tres jugadores. Con este pequeño mazo, el juego termina después de siete parejas encontradas, en vez de veinticinco.

```
>>> import cartas
>>> juego = cartas.JuegoSolterona()
>>> juego.jugar(["Allen","Jeff","Chris"])
---------- Las cartas se han repartido
```

```
Mano Allen contiene
Rey de Corazones
 Jota de Treboles
  Reina de Picas
   Rey de Picas
    10 de Diamantes

Mano Jeff contiene
Reina de Corazones
 Jota de Picas
  Jota de Corazones
   Rey de Diamantes
    Reina de Diamantes

Mano Chris contiene
Jota de Diamantes
 Rey de Treboles
  10 de Picas
   10 de Corazones
    10 de Treboles

Mano Jeff: Reina de Corazones se empareja con Reina de
Diamantes
Mano Chris: 10 de Picas se empareja con 10 de Treboles
----------  Parejas eliminadas, comienza el juego
Mano Allen contiene
Rey de Corazones
 Jota de Treboles
  Reina de Picas
   Rey de Picas
    10 de Diamantes

Mano Jeff contiene
Jota de Picas
 Jota de Corazones
  Rey de Diamantes

Mano Chris contiene
Jota de Diamantes
 Rey de Treboles
  10 de Corazones
```

```
Mano Allen escogio Rey de Diamantes
Mano Allen: Rey de Corazones se empareja con Rey de
Diamantes
Mano Jeff escogio 10 de Corazones
Mano Chris escogio Jota de Treboles
Mano Allen escogio Jota de Corazones
Mano Jeff escogio Jota de Diamantes
Mano Chris escogio Reina de Picas
Mano Allen escogio Jota de Diamantes
Mano Allen: Jota de Corazones se empareja con Jota de
Diamantes
Mano Jeff escogio Rey de Treboles
Mano Chris escogio Rey de Picas
Mano Allen escogio 10 de Corazones
Mano Allen: 10 de Diamantes se empareja con 10 de Corazones
Mano Jeff escogio Reina de Picas
Mano Chris escogio Jota de Picas
Mano Chris: Jota de Treboles se empareja con Jota de Picas
Mano Jeff escogio Rey de Picas
Mano Jeff: Rey de Treboles se empareja con Rey de Picas
---------- Game is Over
Mano Allen esta vacia

Mano Jeff contiene
Reina de Picas

Mano Chris esta vacia
```

Asi que Jeff pierde.

17.8. Glosario

Herencia: es la capacidad de definir una clase, modificando una clase definida previamente.

Clase madre: esta es la clase de la que una clase hereda.

Clase hija: nueva clase creada por medio de herencia, también recibe el nombre de "subclase."

Capítulo 18

Listas enlazadas

18.1. Referencias incrustadas

Hemos visto ejemplos de atributos (denominados **referencias incrustadas**) que se refieren a otros objetos en la sección 13.8. Una estructura de datos muy común (la **lista enlazada**), toma ventaja de esta posibilidad.
Las listas enlazadas están hechas de **nodos**, que contienen una referencia al siguiente nodo en la lista. Además, cada nodo contiene una información denominada la **carga**.
Una lista enlazada se considera como una **estructura de datos recursiva** si damos la siguiente definición.

> Una lista enlazada es:
>
> - la lista vacía, representada por el valor `None`, o
> - un nodo que contiene una carga y una referencia a una lista enlazada.

Las estructuras de datos recursivas se implementan naturalmente con métodos recursivos.

18.2. La clase `Nodo`

Empezaremos con los métodos básicos de inicialización y el `__str__` para que podamos crear y desplegar objetos:

```
class Nodo:
  def __init__(self, carga=None, siguiente=None):
```

```
    self.carga = carga
    self.siguiente  = siguiente

  def __str__(self):
    return str(self.carga)
```

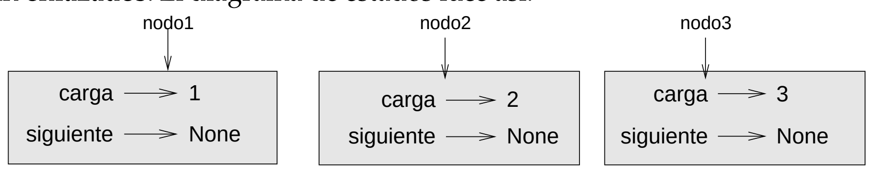
Los parámetros para el método de inicialización son opcionales. Por defecto la carga y el enlace `siguiente`, reciben el valor `None`.
La representación textual de un nodo es la representación de la carga. Como cualquier valor puede ser pasado a la función `str`, podemos almacenar cualquier tipo de valor en la lista.
Para probar la implementación, podemos crear un `Nodo` e imprimirlo:

```
>>> nodo = Nodo("test")
>>> print nodo
test
```

Para hacerlo más interesante, vamos a pensar en una lista con varios nodos:

```
>>> nodo1 = Nodo(1)
>>> nodo2 = Nodo(2)
>>> nodo3 = Nodo(3)
```

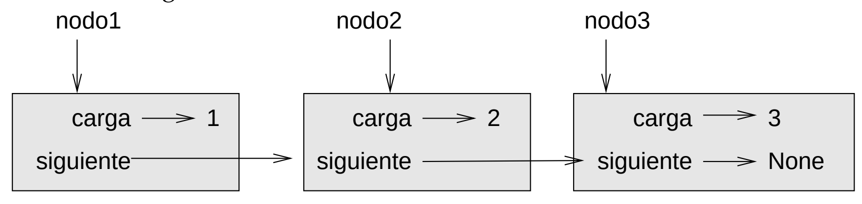
Este código crea tres nodos, pero todavía no tenemos una lista porque estos no estan **enlazados**. El diagrama de estados luce así:



Para enlazar los nodos, tenemos que lograr que el primer nodo se refiera al segundo, y que el segundo se refiera al tercero:

```
>>> nodo1.siguiente = nodo2
>>> nodo2.siguiente = nodo3
```

La referencia del tercer nodo es `None`, lo que indica que es el último nodo de la lista. Ahora el diagrama de estados luce así:



Ahora usted sabe cómo crear nodos y enlazarlos para crear listas. Lo que todavía no está claro, es el por qué hacerlo.

18.3. Listas como colecciones

Las listas son útiles porque proporcionan una forma de ensamblar múltiples objetos en una entidad única, a veces llamada **colección**. En el ejemplo, el primer nodo de la lista sirve como referencia a toda la lista.

Para pasar la lista como parámetro, sólo tenemos que pasar una referencia al primer nodo. Por ejemplo, la función `imprimirLista` toma un solo nodo como argumento. Empieza con la cabeza de la lista, imprime cada nodo hasta llegar al final:

```
def imprimirLista(nodo):
  while nodo:
    print nodo,
    nodo = nodo.siguiente
  print
```

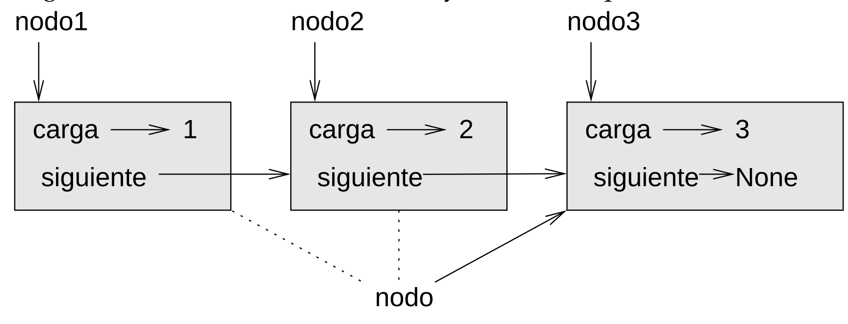
Para llamar este método, pasamos una referencia al primer nodo:

```
>>> imprimirLista(nodo1)
1 2 3
```

Dentro de `imprimirLista` tenemos una referencia al primer nodo de la lista, pero no hay variable que se refiera a los otros nodos. Tenemos que usar el valor `siguiente` de cada nodo para obtener el siguiente nodo.

Para recorrer una lista enlazada, es muy común usar una variable de ciclo como `nodo` para que se refiera a cada uno de los nodos en cada momento.

Este diagrama muestra el valor de `lista` y los valores que `nodo` toma:



Por convención, las listas se imprimen entre corchetes y los elementos se separan por medio de comas, como en el ejemplo `[1, 2, 3]`*. Como ejercicio modifique* `imprimirLista` *de forma que muestre la salida en este formato.*

18.4. Listas y recursión

Es natural implementar muchas operaciones sobre listas por medio de métodos recursivos. Por ejemplo, el siguiente algoritmo recursivo imprime una lista al revés:

1. Separe la lista en dos partes: el primer nodo (la cabeza de la lista); y el resto.
2. Imprima el resto al revés.
3. Imprima la cabeza.

Por supuesto que el paso 2, el llamado recursivo, asume que ya tenemos una forma de imprimir una lista al revés. Si asumimos que esto es así —el salto de fe— entonces podemos convencernos de que el algoritmo trabaja correctamente.
Todo lo que necesitamos es un caso base y una forma de demostrar que, para cualquier lista, eventualmente llegaremos al caso base. Dada la definición recursiva de una lista, un caso base natural es la lista vacía, representada por `None`:

```
def imprimirAlReves(lista):
  if lista == None:
    return
  cabeza = lista
  resto = lista.siguiente
  imprimirAlReves(resto)
  print cabeza,
```

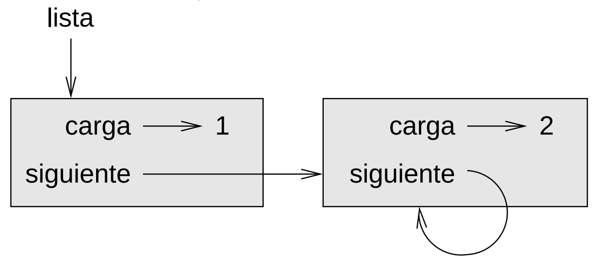
La primera línea resuelve el caso base. Las siguientes separan la `cabeza` y el `resto`. Las últimas dos líneas imprimen la lista. La coma al final de la última línea evita que Python introduzca una nueva línea después de cada nodo.
Ahora llamamos a este método:

```
>>> imprimirAlReves(nodo1)
3 2 1
```

El efecto es una impresión la lista, al revés. Una pregunta natural que usted se puede estar formulando es, ¿por qué razón `imprimirAlReves` e `imprimirLista` son funciones y no métodos en la clase `Nodo`? La razón es que queremos usar a `None` para representar la lista vacía y no se puede llamar un método sobre `None` en Python. Esta limitación hace un poco engorroso escribir el código para manipulación de listas siguiendo la programación orientada a objetos.
¿Podemos demostrar que `imprimirAlReves` va a terminar siempre? En otras palabras, ¿llegará siempre al caso base? De hecho, la respuesta es negativa, algunas listas causarán un error de ejecución.

18.5. Listas infinitas

No hay manera de evitar que un nodo se refiera a un nodo anterior en la lista hacia "atrás". Incluso, puede referirse a sí mismo. Por ejemplo, la siguiente figura muestra una lista con dos nodos, uno de los cuales se refiere a sí mismo:



Si llamamos a `imprimirLista` sobre esta lista, iteraría para siempre. Si llamamos a `imprimirAlReves`, se haría recursión hasta causar un error en tiempo de ejecución. Este comportamiento hace a las listas circulares muy difíciles de manipular.
Sin embargo, a veces son muy útiles. Por ejemplo, podemos representar un número como una lista de dígitos y usar una lista infinita para representar una fracción periódica.
Así que no es posible demostrar que `imprimirLista` e `imprimirAlReves` terminen. Lo mejor que podemos hacer es probar la sentencia, "Si la lista no tiene referencias hacia atrás, los métodos terminarán.". Esto es una **precondición**. Impone una restricción sobre los parámetros y describe el comportamiento del método si ésta se cumple. Más adelante veremos otros ejemplos.

18.6. El teorema de la ambigüedad fundamental

Una parte de `imprimirAlReves` puede haber suscitado su curiosidad:

```
cabeza = lista
resto = lista.siguiente
```

Después de la primera asignación `cabeza` y `lista` tienen el mismo tipo y el mismo valor. ¿Por qué creamos una nueva variable?
La respuesta yace en que las dos variables tienen roles distintos. `cabeza` es una referencia a un nodo y lista es una referencia a toda la lista. Estos "roles" están en la mente del programador y le ayudan a mantener la coherencia de los programas.
En general, no podemos decir inmediatamente qué rol juega una variable en un programa. Esta ambigüedad puede ser útil, pero también dificulta la lectura. Los nombres de las variables pueden usarse para documentar la forma en que esperamos que se use una variable, y, a menudo, podemos crear variables adicionales como `nodo` y `lista` para eliminar ambigüedades.

Podríamos haber escrito `imprimirAlReves` de una manera más concisa sin las variables `cabeza` y `resto`, pero esto también dificulta su lectura:

```
def imprimirAlReves(lista) :
  if lista == None :
     return
  imprimirAlReves(lista.siguiente)
  print lista,
```

Cuando leamos el código, tenemos que recordar que `imprimirAlReves` trata a su argumento como una colección y `print` como a un solo nodo.
El **teorema de la ambigüedad fundamental** describe la ambigüedad inherente en la referencia a un nodo:

> **Una variable que se refiera a un nodo puede tratar el nodo como un objeto único o como el acceso a la lista de nodos**

18.7. Modificando listas

Hay varias formas de modificar una lista enlazada. La obvia consiste en cambiar la carga de uno de sus nodos. Las mas interesantes son las que agregan, eliminan o reordenan los nodos.
Como ejemplo, escribamos un método que elimine el segundo nodo en la lista y retorne una referencia al nodo eliminado

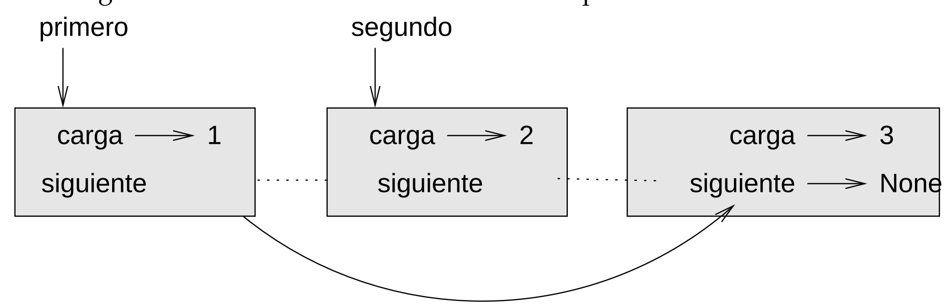
```
def eliminarSegundo(lista):
  if lista == None:
     return
  primero = lista
  segundo = lista.siguiente
  # hacemos que el primer nodo se refiera al tercero
  primero.siguiente = segundo.siguiente
  # desconectamos el segundo nodo de la lista
  segundo.siguiente = None
  return segundo
```

Aquí también estamos usando variables temporales para aumentar la legibilidad. Aquí hay un ejemplo de uso del método:

```
>>> imprimirLista(nodo1)
1 2 3
>>> borrado = eliminarSegundo(nodo1)
>>> imprimirLista(borrado)
```

```
2
>>> imprimirLista(nodo1)
1 3
```

Este diagrama de estado muestra el efecto de la operación:



¿Qué pasa si usted llama este método con una lista que contiene un solo elemento (un **singleton**)? ¿Qué pasa si se llama con la lista vacía como argumento? ¿Hay precondiciones para este método? Si las hay, corríjalo de forma que maneje de manera razonable las violaciones a la precondición.

18.8. Funciones facilitadoras y auxiliares

Es bastante útil dividir las operaciones de listas en dos métodos. Con la impresión al revés podemos ilustrarlo, para desplegar `[3, 2, 1]` en pantalla podemos llamar el método `imprimirAlReves` que desplegará `3, 2,` y llamar otro método para imprimir los corchetes y el primer nodo. Nombrémosla así:

```
def imprimirAlRevesBien(lista):
  print "[",
  if lista != None:
    cabeza = lista
    resto = lista.siguiente
    imprimirAlReves(resto)
    print cabeza,
  print "]",
```

Es conveniente chequear que estos métodos funcionen bien para casos especiales como la lista vacía o una lista con un solo elemento (singleton).
Cuando usamos este método en algún programa, llamamos directamente a la función `imprimirAlRevesBien` para que llame a `imprimirAlReves`. En este sentido, `imprimirAlRevesBien` es una función **facilitadora** (wrapper), que utiliza a la otra, `imprimirAlReves` como función **auxiliar** (helper).

18.9. La clase ListaEnlazada

Hay problemas más sutiles en nuestra implementación de listas que vamos a ilustrar desde los efectos a las causas, a partir de una implementación alternativa exploraremos los problemas que resuelve.
Primero, crearemos una clase nueva llamada `ListaEnlazada`. Tiene como atributos un entero con el número de elementos de la lista y una referencia al primer nodo. Las instancias de `ListaEnlazada` sirven como mecanismo de control de listas compuestas por instancias de la clase `Nodo`:

```
class ListaEnlazada :
  def __init__(self) :
    self.numElementos = 0
    self.cabeza   = None
```

Lo bueno de la clase `ListaEnlazada` es que proporciona un lugar natural para definir las funciones facilitadores como `imprimirAlRevesBien` como métodos:

```
class ListaEnlazada:
  ...
  def imprimirAlReves(self):
    print "[",
    if self.cabeza != None:
      self.cabeza.imprimirAlReves()
    print "]",

class Nodo:
  ...
  def imprimirAlReves(self):
    if self.siguiente != None:
      resto = self.siguiente
      resto.imprimirAlReves()
    print self.carga,
```

Aunque inicialmente pueda parecer un poco confuso, se renombró la función `imprimirAlRevesBien`. Ahora implementamos dos métodos con el mismo nombre `imprimirAlReves`: uno en la clase `Nodo` (el auxiliar); y uno en la clase `ListaEnlazada` (el facilitador). Cuando el facilitador llama al otro método, `self.cabeza.imprimirAlReves`, está invocando al auxiliar, porque `self.cabeza` es una instancia de la clase `Nodo`.
Otro beneficio de la clase `ListaEnlazada` es que facilita agregar o eliminar el primer elemento de una lista. Por ejemplo, `agregarAlPrincipio` es un método de la clase `ListaEnlazada` que toma una carga como argumento y la pone en un nuevo nodo al principio de la lista:

```
class ListaEnlazada:
  ...
  def agregarAlPrincipio(self, carga):
    nodo = Nodo(carga)
    nodo.siguiente = self.cabeza
    self.cabeza = nodo
    self.numElementos = self.numElementos + 1
```

Como de costumbre, usted debe revisar este código para verificar qué sucede con los casos especiales. Por ejemplo, ¿qué pasa si se llama cuando la lista está vacía?

18.10. Invariantes

Algunas listas están "bien formadas". Por ejemplo, si una lista contiene un ciclo, causará problemas graves a nuestros métodos, así que deseamos evitar a toda costa que las listas tengan ciclos. Otro requerimiento de las listas es que el número almacenado en el atributo `numElementos` de la clase `ListaEnlazada` sea igual al número de elementos en la lista.

Estos requerimientos se denominan **Invariantes** porque, idealmente, deberían ser ciertos para todo objeto de la clase en todo momento. Es una muy buena práctica especificar los Invariantes para los objetos porque permite comprobar de manera mas sencilla la corrección del código, revisar la integridad de las estructuras de datos y detectar errores.

Algo que puede confundir acerca de los invariantes es que hay ciertos momentos en que son violados. Por ejemplo, en el medio de `agregarAlPrincipio`, después de que hemos agregado el nodo, pero antes de incrementar el atributo `numElementos`, el Invariante se viola. Esta clase de violación es aceptable, de hecho, casi siempre es imposible modificar un objeto sin violar un Invariante, al menos momentáneamente. Normalmente, requerimos que cada método que viole un invariante, lo establezca nuevamente.

Si hay una parte significativa de código en la que el Invariante se viola, es importante documentarlo claramente, de forma que no se ejecuten operaciones que dependan del Invariante.

18.11. Glosario

Referencia incrustada: referencia almacenada en un atributo de un objeto.

Lista enlazada: es la estructura de datos que implementa una colección por medio de una secuencia de nodos enlazados.

Nodo: elemento de la lista, usualmente implementado como un objeto que contiene una referencia hacia otro objeto del mismo tipo.

Carga: dato contenido en un nodo.

Enlace: referencia incrustada, usada para enlazar un objeto con otro.

Precondición: condición lógica (o aserción) que debe ser cierta para que un método funcione correctamente.

Teorema fundamental de la ambigüedad: la referencia a un nodo de una lista puede interpretarse hacia un nodo determinado o como la referencia a toda la lista de nodos.

Singleton: lista enlazada con un solo nodo.

Facilitador: método que actúa como intermediario entre alguien que llama un método y un método auxiliar. Se crean normalmente para facilitar los llamados y hacerlos menos propensos a errores.

Método auxiliar: es un método que el programador no llama directamente, sino que es usado por otro método para realizar parte de una operación.

Invariante: aserción que debe ser cierta para un objeto en todo momento (excepto cuando el objeto está siendo modificado).

Capítulo 19

Pilas

19.1. Tipos abstractos de datos

Los tipos de datos que ha visto hasta el momento son concretos, en el sentido que hemos especificado completamente como se implementan. Por ejemplo, la clase `Carta` representa una carta por medio de dos enteros. Pero esa no es la única forma de representar una carta; hay muchas representaciones alternativas.
Un **tipo abstracto de datos**, o TAD, especifica un conjunto de operaciones (o métodos) y la semántica de las operaciones (lo que hacen), pero no especifica la implementación de las operaciones. Eso es lo que los hace abstractos.
¿Qué es lo que los hace tan útiles?

- La tarea de especificar un algoritmo se simplifica si se pueden denotar las operaciones sin tener que pensar al mismo tiempo como se implementan.

- Como usualmente hay muchas formas de implementar un TAD, puede ser provechoso escribir un algoritmo que pueda usarse con cualquier implementación alternativa.

- Los TADs bien conocidos, como el TAD Pila de este capítulo, a menudo se encuentran implementados en las bibliotecas estándar de los lenguajes de programación, así que pueden escribirse una sola vez y usarse muchas veces.

- Las operaciones de los TADs nos proporcionan un lenguaje de alto nivel para especificar algoritmos.

Cuando hablamos de TADs hacemos la distinción entre el código que utiliza el TAD, denominado código **cliente**, del código que implementa el TAD, llamado código **proveedor**.

19.2. El TAD Pila

Como ya hemos aprendido a usar otras colecciones como los diccionarios y las listas, en este capítulo exploraremos un TAD muy general, la **pila**.
Una pila es una colección, esto es, una estructura de datos que contiene múltiples elementos.
Un TAD se define por las operaciones que se pueden ejecutar sobre él, lo que recibe el nombre de **interfaz**. La interfaz de una pila comprende las siguientes operaciones:

`__init__`: inicializa una pila vacía.

`meter`: agrega un objeto a la pila.

`sacar`: elimina y retorna un elemento de la pila. El objeto que se retorna siempre es el último que se agregó.

`estaVacia`: revisa si la pila está vacía.

Una pila también se conoce como una estructura"último que Entra, Primero que Sale " o UEPS, porque el último dato que entró es el primero que va a salir.

19.3. Implementando pilas por medio de listas de Python

Las operaciones de listas que Python proporciona son similares a las operaciones que definen una pila. La interfaz no es lo que uno se espera, pero podemos escribir código que traduzca desde el TAD pila a las operaciones primitivas de Python.
Este código se denomina la **implementación** del TAD Pila. En general, una implementación es un conjunto de métodos que satisfacen los requerimientos sintácticos y semánticos de una interfaz.
Aquí hay una implementación del TAD Pila que usa una lista de Python:

```
class Pila :
  def __init__(self) :
    self.items = []

  def meter(self, item) :
    self.items.append(item)

  def sacar(self) :
    return self.items.pop()
```

```
  def estaVacia(self) :
    return (self.items == [])
```

Una objeto Pila contiene un atributo llamado items que es una lista de los objetos que están en la Pila. El método de inicialización le asigna a items la lista vacía.

Para meter un nuevo objeto en la Pila, meter lo pega a items. Para sacar un objeto de la Pila, sacar usa al método pop que proporciona Python para eliminar el último elemento de una lista. Finalmente, para verificar si la Pila está vacía, estaVacia compara a items con la lista vacía.

Una implementación como ésta, en la que los métodos son simples llamados de métodos existentes, se denomina **barniz**. En la vida real, el barniz es una delgada capa de protección que se usa algunas veces en la fabricación de muebles para ocultar la calidad de la madera que recubre. Los científicos de la computación usan esta metáfora para describir una pequeña porción de código que oculta los detalles de una implementación para proporcionar una interfaz más simple o más estandarizada.

19.4. Meter y sacar

Una Pila es una **estructura de datos genérica**, o sea que podemos agregar datos de cualquier tipo a ella. El siguiente ejemplo mete dos enteros y una cadena en la Pila:

```
>>> s = Pila()
>>> s.meter(54)
>>> s.meter(45)
>>> s.meter("+")
```

Podemos usar los métodos estaVacia y sacar para eliminar e imprimir todos los objetos en la Pila:

```
while not s.estaVacia() :
  print s.sacar(),
```

La salida es + 45 54. En otras palabras, acabamos de usar la Pila para imprimir los objetos al revés, ¡y de una manera muy sencilla!

Compare esta porción de código con la implementación de imprimirAlReves en la Sección 18.4. Hay una relación muy profunda e interesante entre la versión recursiva de imprimirAlReves y el ciclo anterior. La diferencia reside en que imprimirAlReves usa la Pila que provee el ambiente de ejecución de Python para llevar pista de los nodos mientras recorre la lista, y luego los imprime cuando la recursión se empieza a devolver. El ciclo anterior hace lo mismo, pero explícitamente por medio de un objeto Pila.

19.5. Evaluando expresiones postfijas con una Pila

En la mayoría de los lenguajes de programación las expresiones matemáticas se escriben con el operador entre los operandos, como en `1+2`. Esta notación se denomina **infija**. Una alternativa que algunas calculadoras usan se denomina notación **postfija**. En la notación postfija, el operador va después de los operandos, como en `1 2 +`.
La razón por la que esta notación es útil reside en que hay una forma muy natural de evaluar una expresión postfija usando una Pila:

- Comenzando por el inicio de la expresión, ir leyendo cada término (operador u operando).
    - Si el término es un operando, meterlo en la Pila.
    - Si el término es un operador, sacar dos operandos de la Pila, ejecutar la operación sobre ellos, y meter el resultado en la Pila.
- Cuando llegue al final de la expresión, tiene que haber un solo aperando en la Pila, ese es el resultado de la expresión.

Como ejercicio, aplique este algoritmo a la expresión `1 2 + 3 *`.

Este ejemplo demuestra una de las ventajas de la notación postfija—no hay necesidad de usar paréntesis para controlar el orden de las operaciones. Para obtener el mismo resultado en notación infija tendríamos que haber escrito `(1 + 2) * 3`.

Como ejercicio, escriba una expresión postfija equivalente a `1 + 2 * 3`.

19.6. Análisis sintáctico

Para implementar el algoritmo anterior, necesitamos recorrer una cadena y separarla en operandos y operadores. Este proceso es un ejemplo de **análisis sintáctico**, y los resultados –los trozos individuales que obtenemos—se denominan **lexemas**. Tal vez recuerde estos conceptos introducidos en el capítulo 2.
Python proporciona el método `split` en los módulos `string` y `re` (expresiones regulares). La función `string.split` parte una cadena en una lista de cadenas usando un caracter como **delimitador**. Por ejemplo:

```
>>> import string
>>> string.split("Ha llegado la hora"," ")
['Ha', 'llegado', 'la', 'hora']
```

En este caso, el delimitador es el caracter espacio, así que la cadena se separa cada vez que se encuentra un espacio.
La función `re.split` es mas poderosa, nos permite especificar una expresión regular en lugar de un delimitador. Una expresión regular es una forma de especificar un conjunto de cadenas. Por ejemplo, `[A-Z]` es el conjunto de todas las letras y `[0-9]` es el conjunto de todos los números. El operador ˆ niega un conjunto, así que `[ˆ0-9]` es el conjunto complemento al de números (todo lo que no es un número), y esto es exactamente lo que deseamos usar para separar una expresión postfija:

```
>>> import re
>>> re.split("([ˆ0-9])", "123+456*/")
['123', '+', '456', '*', '', '/', '']
```

Observe que el orden de los argumentos es diferente al que se usa en la función `string.split`; el delimitador va antes de la cadena.
La lista resultante contiene a los operandos `123` y `456`, y a los operadores `*` y `/`. También incluye dos cadenas vacías que se insertan después de los operandos.

19.7. Evaluando expresiones postfijas

Para evaluar una expresión postfija, utilizaremos el analizador sintáctico de la sección anterior y el algoritmo de la anterior a esa. Por simplicidad, empezamos con un evaluador que solo implementa los operadores `+` y `*`:

```
def evalPostfija(expr):
  import re
  listaLexemas = re.split("([ˆ0-9])", expr)
  Pila = Pila()
  Por lexema in listaLexemas:
    if  lexema == '' or lexema == ' ':
      continue
    if  lexema == '+':
      suma = Pila.sacar() + Pila.sacar()
      Pila.meter(suma)
    elif lexema == '*':
      producto = Pila.sacar() * Pila.sacar()
      Pila.meter(producto)
    else:
```

```
      Pila.meter(int(lexema))
  return Pila.sacar()
```

La primera condición ignora los espacios y las cadenas vacías. Las siguientes dos condiciones detectan los operadores. Asumimos por ahora —intrépidamente—, que cualquier caracter no numérico es un operando.
Verifiquemos la función evaluando la expresión (56+47)*2 en notación postfija:

```
>>> print evalPostfija ("56 47 + 2 *")
206
```

Bien, por ahora.

19.8. Clientes y proveedores

Una de los objetivos fundamentales de un TAD es separar los intereses del proveedor, el que escribe el código que implementa el Tipo Abstracto de Datos, y los del cliente, el que lo usa. El proveedor sólo tiene que preocuparse por que la implementación sea correcta —de acuerdo con la especificación del TAD—y no por el cómo va a ser usado.
Por otro lado, el cliente *asume* que la implementación del TAD es correcta y no se preocupa por los detalles. Cuando usted utiliza los tipos primitivos de Python, se está dando el lujo de pensar como cliente exclusivamente.
Por supuesto, cuando se implementa un TAD, también hay que escribir algún código cliente que permita chequear su funcionamiento. En ese caso, usted asume los dos roles y la labor puede ser un tanto confusa. Hay que concentrarse para llevar la pista del rol que se está jugando en un momento determinado.

19.9. Glosario

Tipo Abstracto de Datos (TAD): Un tipo de datos (casi siempre es una colección de objetos) que se define por medio de un conjunto de operaciones y que puede ser implementado de diferentes formas.

Interfaz: conjunto de operaciones que define al TAD.

Implementación: código que satisface los requerimientos sintácticos y semánticos de una interfaz de un TAD.

Cliente: un programa (o la persona que lo escribió) que usa un TAD.

Proveedor: el código (o la persona que lo escribió) que implementa un TAD.

Barniz: una definición de clase que implementa un TAD con métodos que son llamados a otros métodos, a menudo realizando unas transformaciones previas. El barniz no realiza un trabajo significativo, pero sí mejora o estandariza las interfaces a las que accede el cliente.

Estructura de datos genérica: estructura de datos que puede contener objetos de todo tipo.

Infija: una forma de escribir expresiones matemáticas con los operadores entre los operandos.

Postfija: una forma de escribir expresiones matemáticas con los operadores después de los operandos.

Análisis sintáctico: leer una cadena de carácteres o lexemas y analizar su estructura gramatical.

Lexema: conjunto de carácteres que se considera como una unidad para los propósitos del análisis sintáctico, tal como las palabras del lenguaje natural.

Delimitador: caracter que se usa para separar lexemas, tal como los signos de puntuación en el lenguaje natural.

Capítulo 20

Colas

Este capítulo presenta dos TADs: la cola y la cola de prioridad. En la vida real una **Cola** es una línea de clientes esperando por algún servicio. En la mayoría de los casos, el primer cliente en la línea es el próximo en ser atendido. Sin embargo, hay excepciones. En los aeropuertos, los clientes cuyos vuelos están próximos a partir se atienden, sin importar su posición en la cola. En los supermercados, un cliente cortés puede dejar pasar a alguien que va a pagar unos pocos víveres.

La regla que dictamina quién se atiende a continuación se denomina **política de atención**. La más sencilla se denomina **PEPS**, por la frase "Primero que Entra - Primero que Sale". La política más general es la que implementa una **cola de prioridad**, en la que cada cliente tiene asignada una prioridad y siempre se atiende el cliente con la prioridad más alta, sin importar el orden de llegada. Es la política más general en el sentido de que la prioridad puede asignarse bajo cualquier criterio: la hora de partida de un vuelo, cuántos víveres se van a pagar, o qué tan importante es el cliente. No todas las políticas de atención son "justas," pero la justicia está definida por el que presta el servicio.

El TAD Cola y la cola de prioridad TAD tienen el mismo conjunto de operaciones. La diferencia está en la semántica de ellas: una cola utiliza la política PEPS y una cola de prioridad usa la política de prioridad.

20.1. El TAD Cola

El TAD Cola se define por la siguiente interfaz:

`__init__`: inicializa una Cola vacía.

`meter`: agrega un nuevo objeto a la cola.

`sacar`: elimina y retorna un objeto de la Cola. Entre todos los que están dentro de la cola, el objeto retornado fue el primero en agregarse

`estaVacia`: revisa si la cola está vacía.

20.2. Cola enlazada

Esta primera implementación del TAD Cola se denomina **Cola enlazada** porque está compuesta de objetos `Nodo` enlazados. Aquí está la definición:

```
class Cola:
  def __init__(self):
    self.numElementos = 0
    self.primero = None

  def estaVacia(self):
    return (self.numElementos == 0)

  def meter(self, carga):
    nodo = Nodo(carga)
    nodo.siguiente = None
    if self.primero == None:
      # si esta vacia este nodo sera el primero
      self.primero = nodo
    else:
      # encontrar el ultimo nodo
      ultimo = self.primero
      while ultimo.siguiente: ultimo = ultimo.siguiente
      # pegar el nuevo
      ultimo.siguiente = nodo
    self.numElementos = self.numElementos + 1

  def sacar(self):
    carga = self.primero.carga
    self.primero = self.primero.siguiente
    self.numElementos = self.numElementos - 1
    return carga
```

El método `estaVacia` es idéntico al de la `ListaEnlazada`, `sacar` es quitar el enlace del primer nodo. El método `meter` es un poco más largo.
Si la cola está vacía, le asignamos a `primero` el nuevo nodo.Si tiene elementos, recorremos la lista hasta el último nodo y pegamos el nuevo nodo al final. Podemos detectar si hemos llegado al final de la lista porque el atributo `siguiente` tiene el valor `None`.

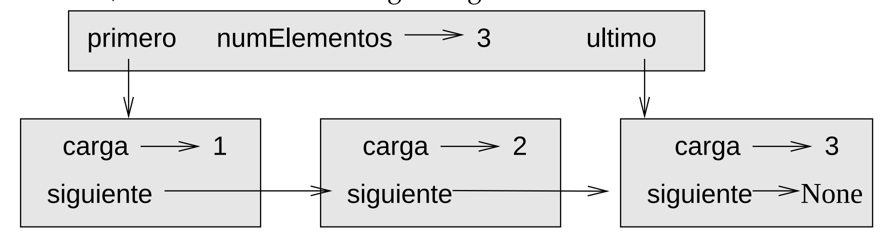
Hay dos invariantes que un objeto `Cola` bien formado debe cumplir. El valor de `numElementos` debe ser el número de nodos en la Cola, y el último nodo debe tener en `siguiente` el valor `None`. Verifique que este método satisfaga los dos invariantes.

20.3. Desempeño

Usualmente, cuando llamamos un método, no nos interesan los detalles de la implementación. Sin embargo, hay un "detalle" que quisiéramos saber —el desempeño del nuevo método. ¿Cuánto tarda en ejecutarse y cómo cambia el tiempo de ejecución a medida que el número de objetos en la Cola crece?
Primero observemos al método `sacar`.
No hay ciclos ni llamados a funciones, así que el tiempo de ejecución de este método es el mismo cada vez que se ejecuta. Los métodos de este tipo se denominan operaciones de **tiempo constante**. De hecho, el método puede ser mas rápido cuando la lista está vacía ya que no entra al cuerpo del condicional, pero esta diferencia no es significativa.
El desempeño de `meter` es muy diferente. En el caso general, tenemos que recorrer la lista para encontrar el último elemento.
Este recorrido toma un tiempo proporcional al atributo numElementos de la lista. Ya que el tiempo de ejecución es una función lineal de numElementos, se dice que este método tiene un tiempo de ejecución de **tiempo lineal**. Comparado con el tiempo constante, es bastante malo.

20.4. Cola Enlazada mejorada

Nos gustaría contar con una implementación del TAD Cola, cuyas operaciones tomen tiempo constante. Una forma de hacerlo es manteniendo una referencia al último nodo, como se ilustra en la figura siguiente:



La implementación de `ColaMejorada` es la siguiente:

```
class ColaMejorada:
  def __init__(self):
```

```
    self.numElementos = 0
    self.primero   = None
    self.ultimo   = None

  def estaVacia(self):
    return (self.numElementos == 0)
```

Hasta aquí el único cambio es al nuevo atributo `ultimo`. Este debe ser usado en los métodos `meter` y `sacar`:

```
class ColaMejorada:
  ...
  def meter(self, carga):
    nodo = nodo(carga)
    nodo.siguiente = None
    if self.numElementos == 0:
      # si está vacía, el nuevo nodo es primero y ultimo
      self.primero = self.ultimo = nodo
    else:
      # encontrar el ultimo nodo
      ultimo = self.ultimo
      # pegar el nuevo nodo
      ultimo.siguiente = nodo
      self.ultimo = nodo
    self.numElementos = self.numElementos + 1
```

Ya que `ultimo` lleva la pista del último nodo, no tenemos que buscarlo. Como resultado, este método tiene un tiempo constante de ejecución.
Hay un precio que pagar por esta mejora. Tenemos que agregar un caso especial a `sacar` que asigne a `ultimo` el valor `None` cuando se saca el único elemento:

```
class ColaMejorada:
  ...
  def sacar(self):
    carga     = self.primero.carga
    self.primero = self.primero.siguiente
    self.numElementos = self.numElementos - 1
    if self.numElementos == 0:
      self.ultimo = None
    return carga
```

Esta implementación es más compleja que la inicial y es más difícil demostrar su corrección. La ventaja es que hemos logrado el objetivo —`meter` y `sacar` son operaciones que se ejecutan en un tiempo constante.

Como ejercicio, escriba una implementación del TAD Cola usando una lista de Python. Compare el desempeño de esta implementación con el de la `ColaMejorada` *para un distintos valores de numElementos.*

20.5. Cola de prioridad

El TAD cola de prioridad tiene la misma interfaz que el TAD Cola, pero su semántica es distinta.

`__init__`: inicializa una Cola vacía.

`meter`: agrega un objeto a la Cola.

`sacar`: saca y retorna un objeto de la Cola. El objeto que se retorna es el de la mas alta prioridad.

`estaVacia`: verifica si la Cola está vacía.

La diferencia semántica está en el objeto que se saca, que necesariamente no es el primero que se agregó. En vez de esto, es el que tiene el mayor valor de prioridad. Las prioridades y la manera de compararlas no se especifica en la implementación de la cola de prioridad. Depende de cuáles objetos estén en la Cola.
Por ejemplo, si los objetos en la Cola tienen nombres, podríamos escogerlos en orden alfabético. Si son puntajes de bolos iríamos sacando del más alto al más bajo; pero si son puntajes de golf, iríamos del más bajo al más alto. En tanto que podamos comparar los objetos en la Cola, podemos encontrar y sacar el que tenga la prioridad más alta.
Esta implementación de la cola de prioridad tiene como atributo una lista de Python que contiene los elementos en la Cola.

```
class ColaPrioridad:
  def __init__(self):
    self.items = []

  def estaVacia(self):
    return self.items == []

  def meter(self, item):
    self.items.append(item)
```

El método de inicialización, `estaVacia`, y `meter` solo son barniz para operaciones sobre listas. El único interesante es `sacar`:

```
class ColaPrioridad:
  ...
  def sacar(self):
    maxi = 0
    for i in range(1,len(self.items)):
      if self.items[i] > self.items[maxi]:
        maxi = i
    item = self.items[maxi]
    self.items[maxi:maxi+1] = []
    return item
```

Al iniciar cada iteración, `maxi` almacena el índice del ítem más grande (con la prioridad más alta) que hayamos encontrado *hasta el momento*. En cada iteración, el programa compara el `i`ésimo ítem con el que iba ganando. Si el nuevo es mejor, el valor de `maxi` se actualiza con el de `i`.

Cuando el `for` se completa, `maxi` es el índice con el mayor ítem de todos. Éste se saca de la lista y se retorna.

Probemos la implementación:

```
>>> q = ColaPrioridad()
>>> q.meter(11)
>>> q.meter(12)
>>> q.meter(14)
>>> q.meter(13)
>>> while not q.estaVacia(): print q.sacar()
14
13
12
11
```

Si la Cola contiene números o cadenas, se sacan en orden alfabético o numérico, del más alto al más bajo. Python puede encontrar el mayor entero o cadena a través de los operadores de comparación primitivos.

Si la Cola contiene otro tipo de objeto, creado por el programador, tiene que proporcionar el método `__cmp__`. Cuando `sacar` use al operador > para comparar ítems, estaría llamando el método `__cmp__` sobre el primero y pasándole al segundo como parámetro. En tanto que `__cmp__` funcione correctamente, la cola de prioridad será correcta.

20.6. La clase `golfista`

Un ejemplo poco usado de definición de prioridad es la clase `golfista` que lleva el registro de los nombres y los puntajes de jugadores de golf. Primero definimos

__init__ y __str__:

```
class golfista:
  def __init__(self, nombre, puntaje):
    self.nombre = nombre
    self.puntaje= puntaje

  def __str__(self):
    return "%-16s: %d" % (self.nombre, self.puntaje)
```

__str__ utiliza el operador de formato para poner los nombres y los puntajes en dos columnas.

A continuación definimos una versión de __cmp__ en la que el puntaje mas bajo tenga la prioridad mas alta. Recuerde que para Python __cmp__ retorna 1 si self es "mayor que" otro, -1 si self es "menor" otro, y 0 si son iguales.

```
class golfista:
  ...
  def __cmp__(self, otro):
    # el menor tiene mayor prioridad
    if self.puntaje < otro.puntaje: return  1
    if self.puntaje > otro.puntaje: return -1
    return 0
```

Ahora estamos listos para probar la cola de prioridad almacenando instancias de la clase golfista:

```
>>> tiger = golfista("Tiger Woods",    61)
>>> phil  = golfista("Phil Mickelson", 72)
>>> hal   = golfista("Hal Sutton",     69)
>>>
>>> pq = ColaPrioridad()
>>> pq.meter(tiger)
>>> pq.meter(phil)
>>> pq.meter(hal)
>>> while not pq.estaVacia(): print pq.sacar()
Tiger Woods    : 61
Hal Sutton     : 69
Phil Mickelson : 72
```

Como ejercicio, escriba una implementación del TAD cola de prioridad TAD usando una lista enlazada. Ésta debe mantenerse ordenada, de forma que sacar sea una operación de tiempo constante. Compare el desempeño de esta implementación con la implementación basada en listas de Python.

20.7. Glosario

Cola: conjunto ordenado de objetos (o personas) esperando a que se les preste algún servicio

Cola: TAD con las operaciones que se realizan en una Cola.

Política de atención: reglas que determinan cuál es el siguiente objeto que se saca (atiende) en una Cola.

PEPS: "Primero que Entra, Primero que Sale" , política de atención en la que se saca el primer elemento de la Cola.

Atención por prioridad: una política de atención en la que se saca el elemento de la Cola que tenga la mayor prioridad.

Cola de prioridad: un TAD que define las operaciones que se pueden realizar en una cola de prioridad.

Cola enlazada: implementación de una Cola que utiliza una lista enlazada.

Desempeño: toda función de un TAD realiza un número de operaciones básicas que dependen del número de elementos que éste contiene en un momento dado. Por medio de este número de operaciones básicas se pueden comparar distintas alternativas de implementación de una operación.

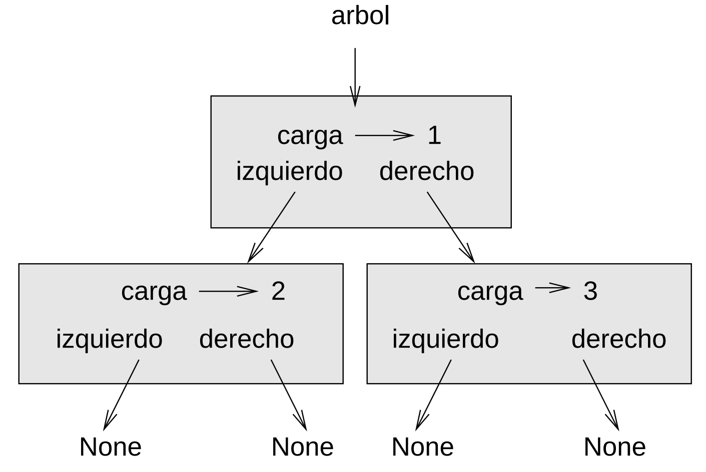
Tiempo constante: desempeño de una operación, cuyo tiempo de ejecución no depende del tamaño de la estructura de datos.

Tiempo lineal: desempeño de una operación, cuyo tiempo de ejecución es una función lineal del tamaño de la estructura de datos.

Capítulo 21

Árboles

Como las listas enlazadas, los árboles están compuestos de nodos. Una clase muy común es el **árbol binario**, en el que cada nodo contiene referencias a otros dos nodos (posiblemente, valores None). Estas referencias se denominan los subárboles izquierdo y derecho. Como los nodos de las listas, los nodos de los árboles también contienen una carga. Un diagrama de estados para los árboles luce así:



Para evitar el caos en las figuras, a menudo omitimos los valores `None`.

El inicio del árbol (al nodo al que `árbol` se refiere) se denomina **raíz**. Para conservar la metáfora con los árboles, los otros nodos se denominan ramas, y los nodos que tienen referencias nulas se llaman **hojas**. Parece extraño el dibujo con la raíz en la parte superior y las hojas en la inferior, pero esto es sólo el principio.

Los científicos de la computación también usan otra metáfora—el árbol genealógico. El nodo raíz se denomina **padre** y los nodos a los que se refiere **hijos**, los nodos

que tienen el mismo padre se denominan **hermanos**.
Finalmente, hay un vocabulario geométrico para referirse a los árboles. Ya mencionamos la distinción entre izquierda y derecha, también se acostumbra diferenciar entre "arriba" (hacia el padre/raíz) y "abajo" (hacia los hijos/hojas). Además, todos los nodos que están a una misma distancia de la raíz comprenden un **nivel**.
Probablemente no necesitemos estas metáforas para describir los árboles, pero se usan extensivamente.
Como las listas enlazadas, los árboles son estructuras de datos recursivas ya que su definición es recursiva.

> Un árbol es:
>
> - el árbol vacío, representado por `None`, o
> - Un nodo que contiene una referencia a un objeto y referencias a otros aŕboles.

21.1. Construyendo árboles

El proceso de construir un árbol es similar al de una lista enlazada. La llamada al constructor arma un árbol con un solo nodo.

```
class arbol:
  def __init__(self, carga, izquierdo=None, derecho=None):
    self.carga = carga
    self.izquierdo  = izquierdo
    self.derecho = derecho

  def __str__(self):
    return str(self.carga)
```

La `carga` puede tener cualquier tipo, pero los parámetros `izquierdo` y `derecho` deben ser nodos. En `__init__`, `izquierdo` y `derecho` son opcionales; su valor por defecto es `None`. Imprimir un nodo equivale a imprimir su carga.
Una forma de construir un árbol es de abajo hacia arriba. Primero se construyen los nodos hijos:

```
izquierdo = arbol(2)
derecho = arbol(3)
```

Ahora se crea el padre y se enlazan los hijos:

```
a = arbol(1, izquierdo, derecho);
```

Podemos escribir esto de una manera más compacta anidando los llamados:

```
>>> a = arbol(1, arbol(2), arbol(3))
```

Con las dos formas se obtiene como resultado el árbol que ilustramos al principio del capítulo.

21.2. Recorridos sobre árboles

Cada vez que se encuentre con una nueva estructura de datos su primera pregunta debería ser, "¿Cóomo la recorro?". La forma más natural de recorrer un árbol es recursiva. Si el árbol contiene números enteros en la carga, esta función calcula su suma :

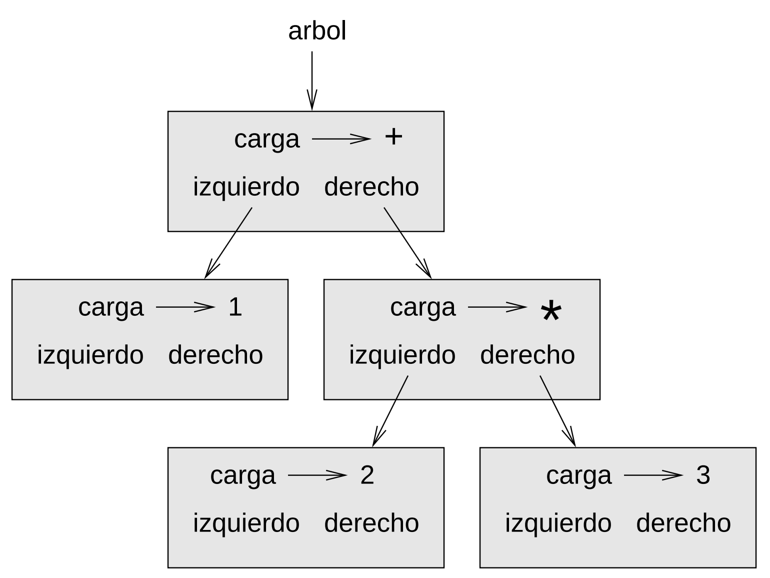
```
def total(a):
  if a == None:
     return 0
  else:
     return total(a.izquierdo) + total(a.derecho)
            + a.carga
```

El caso base se da cuando el argumento es el árbol vacío, que no tiene carga, así que la suma se define como 0. El paso recursivo realiza dos llamados recursivos para encontrar la suma de los árboles hijos, cuando finalizan, se suma a estos valores la carga del padre.

21.3. Árboles de expresiones

Un árbol representa naturalmente la estructura de una expresión. Además, lo puede realizar sin ninguna ambigüedad. Por ejemplo, la expresión infija `1 + 2 * 3` es ambigua a menos que se establezca que la multiplicación se debe realizar antes que la suma.

Este árbol representa la misma expresión:



Los nodos de un árbol para una expresión pueden ser operandos como `1` y `2`, también operadores como `+` y `*`. Los operandos deben ser nodos hoja; y los nodos que contienen operadores tienen referencias a sus operandos. Todos estos operadores son **binarios**, así que solamente tienen dos operandos.
Un árbol como el siguiente representa la figura anterior:

```
>>> a = arbol('+', arbol(1),
                       arbol('*', arbol(2), arbol(3)))
```

Observando la figura no hay ninguna duda sobre el orden de las operaciones; la multiplicación ocurre primero para que se calcule el segundo operando de la suma.
Los árboles de expresiones tienen muchos usos. El ejemplo de este capítulo utiliza árboles para traducir expresiones entre las notaciones postfija, prefija, e infija. Árboles similares se usan en los compiladores para analizar sintácticamente, optimizar y traducir programas.

21.4. Recorrido en árboles

Podemos recorrer un árbol de expresiones e imprimir el contenido de la siguiente forma:

```
def imprimirarbol(a):
  if a == None:
     return
```

```
  print a.carga,
  imprimirarbol(a.izquierdo)
  imprimirarbol(a.derecho)
```

En otras palabras, para imprimir un árbol, primero se imprime el contenido (carga) de la raíz, luego todo el subárbol izquierdo, y a continuación todo el subárbol derecho. Este recorrido se denomina **preorden**, porque el contenido de la raíz se despliega *antes* que el contenido de los hijos. Para el ejemplo anterior, la salida es:

```
>>> a = arbol('+', arbol(1),
                       arbol('*', arbol(2), arbol(3)))
>>> imprimirarbol(a)
+ 1 * 2 3
```

Esta notación diferente a la infija y a la postfija, se denomina **prefija**, porque los operadores aparecen antes que sus operandos.
Usted puede sospechar que si se recorre el árbol de una forma distinta se obtiene otra notación. Por ejemplo si se despliegan los dos subárboles primero y a continuación el nodo raíz, se obtiene

```
def imprimirarbolPostorden(a):
  if a == None:
     return
  else
     imprimirarbolPostorden(a.izquierdo)
     imprimirarbolPostorden(a.derecho)
     print a.carga,
```

El resultado, `1 2 3 * +`, está en notación postfija!. Por esta razón este recorrido se denomina **postorden**.
Finalmente, para recorrer el árbol **en orden**, se imprime el árbol izquierdo, luego la raíz y, por último, el árbol derecho:

```
def imprimirabolEnOrden(a):
  if a == None:
     return
  imprimirabolEnOrden(a.izquierdo)
  print a.carga,
  imprimirabolEnOrden(a.derecho)
```

El resultado es `1 + 2 * 3`, la expresión en notación infija.
Por precisión debemos anotar que hemos omitido una complicación importante. Algunas veces cuando escribimos una expresión infija, tenemos que usar paréntesis para preservar el orden de las operaciones. Así que un recorrido en orden no es suficiente en todos los casos para generar una expresión infija.

Sin embargo, con unas mejoras adicionales, los árboles de expresiones y los tres recorridos recursivos proporcionan una forma general de traducir expresiones de un formato al otro.

> *Como ejercicio, modifique* `imprimirarbolEnOrden` *para que despliegue paréntesis alrededor de cada operador y pareja de operandos. ¿La salida es correcta e inequívoca? ¿Siempre son necesarios los paréntesis?*

Si realizamos un recorrido en orden y llevamos pista del nivel en el que vamos podemos generar una representación gráfica del árbol:

```
def imprimirarbolSangrado(a, nivel=0):
  if a == None:
     return
  imprimirarbolSangrado(a.derecho, nivel+1)
  print '  '*nivel + str(a.carga)
  imprimirarbolSangrado(a.izquierdo, nivel+1)
```

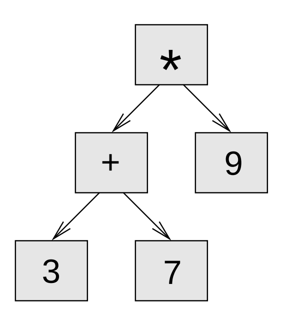
El parámetro `nivel` lleva el nivel actual. Por defecto es 0. Cada vez que hacemos un llamado recursivo pasamos `nivel+1`, porque el nivel de los hijos siempre es uno más del nivel del padre. Cada objeto se sangra o indenta con dos espacios por nivel. El resultado para el árbol de ejemplo es:

```
>>> imprimirarbolSangrado(a)
    3
  *
    2
+
  1
```

Si rota el libro 90 grados en el sentido de las manecillas del reloj verá una forma simplificada del dibujo al principio del capítulo.

21.5. Construyendo un árbol para una expresión

En esta sección, analizaremos sintácticamente expresiones infijas para construir su respectivo árbol de expresión. Por ejemplo, la expresión `(3+7)*9` se representa con el siguiente árbol:



Note que hemos simplificado el diagrama ocultando los nombres de los atributos. El análisis sintáctico se hará sobre expresiones que incluyan números, paréntesis, y los operadores `+` y `*`. Asumimos que la cadena de entrada ha sido separada en una lista de lexemas; por ejemplo, para `(3+7)*9` la lista de lexemas es:

```
['(', 3, '+', 7, ')', '*', 9, 'fin']
```

La cadena `fin` sirve para prevenir que el analizador sintáctico siga leyendo más allá del final de la lista.

> *Como ejercicio escriba una función que reciba una cadena de texto con una expresión y retorne la lista de lexemas (con la cadena* `fin` *al final).*

La primera función que escribiremos es `obtenerLexema`, que toma una lista de lexemas y un lexema esperado como parámetros. Compara el lexema esperado con el primero de la lista: si son iguales, elimina el lexema de la lista y retorna True, si no son iguales, retorna False:

```
def obtenerLexema(listaLexemas, esperado):
  if listaLexemas[0] == esperado:
    del listaLexemas[0]
    return 1
  else:
    return 0
```

Como `listaLexemas` se refiere a un objeto mutable, los cambios que hacemos son visibles en cualquier otra parte del programa que tenga una referencia a la lista.

La siguiente función, `obtenerNumero`, acepta operandos. Si el siguiente lexema en `listaLexemas` es un número, `obtenerNumero` lo elimina y retorna un nodo hoja cuya carga será el número; si no es un número retorna `None`.

```
def obtenerNumero(listaLexemas):
  x = listaLexemas[0]
  if type(x) != type(0):
```

```
    return None
  del listaLexemas[0]
  return arbol (x, None, None)
```

Probemos a `obtenerNumero` con una lista pequeña de números. Después del llamado, imprimimos el árbol resultante y lo que queda de la lista:

```
>>> listaLexemas = [9, 11, 'fin']
>>> x = obtenerNumero(listaLexemas)
>>> imprimirarbolPostorden(x)
9
>>> print listaLexemas
[11, 'fin']
```

El siguiente método que necesitamos es `obtenerProducto`, que construye un árbol de expresión para productos. Un producto sencillo tiene dos números como operandos, como en `3 * 7`.

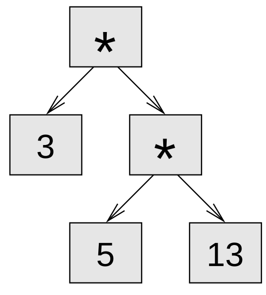
```
def obtenerProducto(listaLexemas):
  a = obtenerNumero(listaLexemas)
  if obtenerLexema(listaLexemas, '*'):
    b = obtenerNumero(listaLexemas)
    return arbol ('*', a, b)
  else:
    return a
```

Asumiendo que `obtenerNumero` retorna un árbol, le asignamos el primer operando a `a`. Si el siguiente carácter es `*`, obtenemos el segundo número y construimos un árbol de expresión con `a`, `b`, y el operador.
Si el siguiente carácter es cualquier otro, retornamos el nodo hoja con `a`. Aquí hay dos ejemplos:

```
>>> listaLexemas = [9, '*', 11, 'fin']
>>> a = obtenerProducto(listaLexemas)
>>> imprimirarbolPostorden(a)
9 11 *


>>> listaLexemas = [9, '+', 11, 'fin']
>>> arbol = obtenerProducto(listaLexemas)
>>> imprimirarbolPostorden(arbol)
9
```

El segundo ejemplo implica que consideramos que un solo operando sea tratado como una clase de producto. Esta definición de “producto” es contraintuitiva, pero resulta ser muy provechosa.
Ahora, tenemos que manejar los productos compuestos, como `3 * 5 * 13`. Esta expresión es un producto de productos, vista así: `3 * (5 * 13)`. El árbol resultante es:



Con un pequeño cambio en `obtenerProducto`, podemos analizar un producto arbitrariamente largo:

```
def obtenerProducto(listaLexemas):
  a = obtenerNumero(listaLexemas)
  if obtenerLexema(listaLexemas, '*'):
    b = obtenerProducto(listaLexemas)   # esta línea cambió
    return arbol ('*', a, b)
  else:
    return a
```

En otras palabras, un producto puede ser un árbol singular o un árbol con `*` en la raíz, un número en el subárbol izquierdo, y un producto en el subárbol derecho. Esta clase de definición recursiva debería empezar a ser familiar.
Probemos la nueva versión con un producto compuesto:

```
>>> listaLexemas = [2, '*', 3, '*', 5 , '*', 7, 'fin']
>>> a = obtenerProducto(listaLexemas)
>>> imprimirarbolPostorden(arbol)
2 3 5 7 * * *
```

Ahora, agregaremos la posibilidad de analizar sumas. Otra vez daremos una definición contraintuitiva de la “suma.” Una suma puede ser un árbol con `+` en la raíz, un producto en el subárbol izquierdo y una suma en el subárbol derecho. O, una suma puede ser sólo un producto.
Si usted analiza esta definición encontrará que tiene una propiedad muy bonita: podemos representar cualquier expresión (sin paréntesis) como una suma de productos. Esta propiedad es el fundamento de nuestro algoritmo de análisis sintáctico.

`obtenerSuma` intenta construir un árbol con un producto en izquierdo y una suma en derecho. Pero si no encuentra un `+`, solamente construye un producto.

```
def obtenerSuma(listaLexemas):
  a = obtenerProducto(listaLexemas)
  if obtenerLexema(listaLexemas, '+'):
    b = obtenerSuma(listaLexemas)
    return arbol ('+', a, b)
  else:
    return a
```

Probemos con `9 * 11 + 5 * 7`:

```
>>> listaLexemas = [9, '*', 11, '+', 5, '*', 7, 'fin']
>>> a = obtenerSuma(listaLexemas)
>>> imprimirarbolPostorden(árbol)
9 11 * 5 7 * +
```

Casi terminamos, pero todavía faltan los paréntesis. En cualquier posición de una expresión donde podamos encontrar un número puede también haber una suma completa cerrada entre paréntesis. Necesitamos modificar `obtenerNumero` para que sea capaz de manejar **subexpresiones**:

```
def obtenerNumero(listaLexemas):
  if obtenerLexema(listaLexemas, '('):
    # obtiene la subexpresión
    x = obtenerSuma(listaLexemas)
    # elimina los paréntesis
    obtenerLexema(listaLexemas, ')')
    return x
  else:
    x = listaLexemas[0]
    if type(x) != type(0):
       return None
    listaLexemas[0:1] = []
    return árbol (x, None, None)
```

Probemos esto con `9 * (11 + 5) * 7`:

```
>>> listaLexemas = [9, '*', '(', 11, '+', 5, ')','*', 7,
'fin']
>>> arbol = obtenerSuma(listaLexemas)
>>> imprimirarbolPostorden(arbol)
9 11 5 + 7 * *
```

El analizador manejó los paréntesis correctamente, la suma se hace antes que la multiplicación.
En la versión final del programa, sería bueno nombrar a `obtenerNumero` con un rol más descriptivo.

21.6. Manejo de errores

En todo momento hemos asumido que las expresiones están bien formadas. Por ejemplo, cuando llegamos al final de una subexpresión, asumimos que el siguiente carácter es un paréntesis derecho. Si hay un error y el siguiente carácter es algo distinto debemos manejar esta situación.

```
def obtenerNumero(listaLexemas):
  if obtenerLexema(listaLexemas, '('):
    x = obtenerSuma(listaLexemas)
    if not obtenerLexema(listaLexemas, ')'):
      raise 'ErrorExpresionMalFormada', 'falta paréntesis'
    return x
  else:
    # el resto del código se omite
```

La sentencia `raise` crea una excepción. En este caso creamos una nueva clase de excepción llamada `ErrorExpresionMalFormada`. Si la función que llamó a `obtenerNumero`, o una de las funciones en la traza causante de su llamado maneja la excepción, el programa puede continuar. De otra forma, Python imprimirá un mensaje de error y abortará la ejecución.

> *Como ejercicio, encuentre otros lugares donde pueden ocurrir errores de este tipo y agregue sentencias* `raise` *apropiadas. Pruebe su código con expresiones mal formadas.*

21.7. El árbol de animales

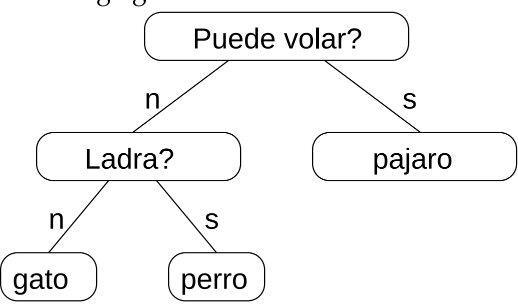
En esta sección desarrollaremos un pequeño programa que usa un árbol para representar una base de conocimiento.
El programa interactúa con el usuario para crear un árbol de preguntas y nombres de animales. Aquí hay una ejecución de ejemplo:

```
¿Esta pensando en un animal? s
¿Es un pájaro? n
¿Cual es el nombre del animal? perro
¿Que pregunta permite distinguir entre un perro
y un pájaro? Puede volar
¿Si el animal fuera perro la respuesta sería? n

¿Esta pensando en un animal? s
¿Puede volar? n
¿Es un perro? n
¿Cual es el nombre del animal? gato
¿Que pregunta permite distinguir un gato
de un perro? Ladra
¿Si el animal fuera un gato
la respuesta sería? n

¿Esta pensando en un animal? y
¿Puede volar? n
¿Ladra? s
¿Es un perro? s
¡Soy el mejor!
```

Este es el árbol que el diálogo genera:



Al principio de cada ronda, el programa empieza en la raíz del árbol y hace la primera pregunta. Dependiendo de la respuesta se mueve al subárbol izquierdo o derecho y continúa hasta que llega a un nodo hoja. En ese momento conjetura. Si falla, le pregunta al usuario el nombre del animal y una pregunta que le permitiría distinguir el animal conjeturado del real. Con esta información agrega un nodo al árbol con la nueva pregunta y el nuevo animal.

Aquí está el código fuente:

```
def animal():
  # Un solo nodo
  raiz = arbol("pajaro")

  # Hasta que el usuario salga
  while True:
    print
    if not si("Esta pensando en un animal? "):
       break

    # Recorrer el arbol
    arbol = raiz
    while arbol.obtenerizquierdo() != None:
      pregunta = arbol.obtenercarga() + "? "
      if si(pregunta):
        árbol = arbol.obtenerderecho()
      else:
        arbol = arbol.obtenerizquierdo()

    # conjetura
    conjetura = arbol.obtenercarga()
    pregunta = "¿Es un" + conjetura + "? "
    if si(pregunta):
      print "¡Soy el mejor!"
      continue

    # obtener mas informacion
    pregunta  = "¿Cual es el nombre el animal? "
    animal  = raw_input(pregunta)
    pregunta  = "¿Que pregunta permitiria
                 distinguir un %s de un %s? "
    q = raw_input(pregunta % (animal,conjetura))

    # agrega un nuevo nodo arbol
    arbol.asignarcarga(q)
    pregunta = "¿Si el animal fuera %s
               la respuesta sería? "
    if si(pregunta % animal):
      arbol.asignarizquierdo(arbol(conjetura))
      árbol.asignarderecho(arbol(animal))
    else:
      arbol.asignarizquierdo(arbol(animal))
```

```
    arbol.asignarderecho(arbol(conjetura))
```

La función `si` es auxiliar, imprime una pregunta y recibe la respuesta del usuario. Si la respuesta empieza con *s* o *S*, retorna cierto:

```
def si(preg):
  from string import lower
  r = lower(raw_input(preg))
  return (r[0] == 's')
```

La condición del ciclo es `True`, lo que implica que el ciclo iterará hasta que la sentencia `break` se ejecute cuando el usuario deje de pensar en animales.
El ciclo `while` interno recorre el árbol desde la raíz hasta el fondo, guiándose por las respuestas del usuario.
Cuando se agrega un nuevo nodo al árbol, la nueva pregunta reemplaza la carga, y los dos hijos son el animal nuevo y la carga original.
Una limitación seria de este programa es que cuando finaliza, ¡olvida todo lo que se le ha enseñado!

> *Como ejercicio, piense en diferentes maneras de guardar este árbol de conocimiento en un archivo. Implemente la que parezca más sencilla.*

21.8. Glosario

Arbol binario: un árbol en el que cada nodo se refiere a cero, uno o dos nodos, llamados hijos.

Raíz: nodo inicial en un árbol, es el único que no tiene padre.

Hoja: nodo que no tiene hijos y se encuentra lo mas abajo posible.

Padre: nodo que tiene la referencia hacia otro nodo.

Hijo: uno de los nodos referenciados por un nodo.

Hermanos: nodos que comparten un padre común.

Nivel: conjunto de nodos equidistantes a la raíz.

Operador binario: operador que acepta dos operandos.

Subexpresión: expresión en paréntesis que se puede ver como un solo operando en una expresión más grande.

Preorden: es el recorrido sobre un árbol en el que se visita cada nodo antes que a sus hijos.

Notación prefija: notación para escribir una expresión matemática en la que cada operador aparece antes de sus operandos.

Postorden: forma de recorrer un árbol, visitando los hijos de un nodo antes que a este.

En orden: forma de recorrer un árbol, visitando primero el subárbol izquierdo, luego la raíz y por último, el subárbol derecho.

Apéndice A

Depuración

Hay diferentes tipos de error que pueden suceder en un programa y es muy útil distinguirlos a fin de rastrearlos más rápidamente:

- Los errores sintácticos se producen cuando Python traduce el código fuente en código objeto. Usualmente indican que hay algún problema en la sintaxis del programa. Por ejemplo, omitir los puntos seguidos al final de una sentencia `def` produce un mensaje de error un poco redundante `SyntaxError: invalid syntax`.
- Los errores en tiempo de ejecución se producen por el sistema de ejecución, si algo va mal mientras el programa corre o se ejecuta. La mayoría de errores en tiempo de ejecución incluyen información sobre la localización del error y las funciones que se estaban ejecutando. Ejemplo: una recursión infinita eventualmente causa un error en tiempo de ejecución de "maximum recursion depth exceeded."
- Los errores semánticos se dan en programas que compilan y se ejecutan normalmente, pero no hacen lo que se pretendía. Ejemplo: una expresión podría evaluarse en un orden inesperado, produciendo un resultado incorrecto.

El primer paso en la depuración consiste en determinar la clase de error con la que se está tratando. Aunque las siguientes secciones se organizan por tipo de error, algunas técnicas se aplican en más de una situación.

A.1. Errores sintácticos

Los errores sintácticos se corrigen fácilmente una vez que usted ha determinado a qué apuntan. Desafortunadamente, en algunas ocasiones los mensajes de error

no son de mucha ayuda. Los mensajes de error más comunes son `SyntaxError: invalid syntax` y `SyntaxError: invalid token`, que no son muy informativos.

Por otro lado, el mensaje sí dice dónde ocurre el problema en el programa. Más precisamente, dice dónde fue que Python encontró un problema, que no necesariamente es el lugar dónde está el error. Algunas veces el error está antes de la localización que da el mensaje de error, a menudo en la línea anterior.

Si usted está construyendo los programas incrementalmente, debería tener una buena idea de dónde se localiza el error. Estará en la última línea que se agregó.

Si usted está copiando código desde un libro, comience por comparar su código y el del libro muy cuidadosamente. Chequee cada carácter. Al mismo tiempo, recuerde que el libro puede tener errores, así que si encuentra algo que parece un error sintáctico, entonces debe serlo.

Aquí hay algunas formas de evitar los errores sintácticos más comunes:

1. Asegúrese de no usar una palabra reservada de Python como nombre de variable

2. Chequee que haya colocado dos puntos seguidos al final de la cabecera de cada sentencia compuesta, incluyendo los ciclos `for`, `while`, los condicionales `if`, las definiciones de función `def` y las `clases`.

3. Chequee que la indentación o sangrado sea consistente. Se puede indentar con espacios o tabuladores, pero es mejor no mezclarlos. Cada nivel debe sangrarse la misma cantidad de espacios o tabuladores.

4. Asegúrese de que las cadenas en los programas estén encerradas entre comillas.

5. Si usted tiene cadenas multilínea creadas con tres comillas consecutivas, verifique su terminación. Una cadena no terminada puede causar un error denominado `invalid token` al final de su programa, o puede tratar la siguiente parte del programa como si fuera una cadena, hasta toparse con la siguiente cadena. En el segundo caso, ¡puede que Python no produzca ningún mensaje de error!

6. Un paréntesis sin cerrar—`(`, `{`, o `[`—hace que Python continúe con la siguiente línea como si fuera parte de la sentencia actual. Generalmente, esto causa un error inmediato en la siguiente línea.

7. Busque por la confusión clásica entre `=` y `==`, adentro y afuera de los condicionales.

Si nada de esto funciona, avance a la siguiente sección...

A.1.1. No puedo ejecutar mi programa sin importar lo que haga

Si el compilador dice que hay un error y usted no lo ha visto, eso puede darse porque usted y el compilador no están observando el mismo código. Chequee su ambiente de programación para asegurarse de que el programa que está editando es el que Python está tratando de ejecutar. Si no está seguro, intente introducir deliberadamente un error sintáctico obvio al principio del programa. Ahora ejecútelo o impórtelo de nuevo. Si el compilador no encuentra el nuevo error, probablemente hay algún problema de configuración de su ambiente de programación.
Si esto pasa, una posible salida es empezar de nuevo con un programa como "Hola todo el mundo!," y asegurarse de que pueda ejecutarlo correctamente. Después, añadir gradualmente a éste los trozos de su programa.

A.2. Errores en tiempo de ejecución

Cuando su programa está bien sintácticamente, Python puede importarlo y empezar a ejecutarlo. ¿Qué podría ir mal ahora?

A.2.1. Mi programa no hace absolutamente nada

Este problema es el más común cuando su archivo comprende funciones y clases pero no hace ningún llamado para empezar la ejecución. Esto puede ser intencional si usted sólo planea importar este módulo para proporcionar funciones y clases.
Si no es intencional, asegúrese de que está llamando a una función para empezar la ejecución, o ejecute alguna desde el indicador de entrada (prompt). Revise la sección posterior sobre el "Flujo de Ejecución".

A.2.2. Mi programa se detiene

Si un programa se para y parece que no está haciendo nada, decimos que está "detenido." Esto a veces sucede porque está atrapado en un ciclo infinito o en una recursión infinita.

- Si hay un ciclo sospechoso de ser la causa del problema, añada una sentencia `print` inmediatamente antes del ciclo que diga "entrando al ciclo" y otra inmediatamente después, que diga "saliendo del ciclo."

  Ejecute el programa. Si obtiene el primer mensaje y no obtiene el segundo, ha encontrado su ciclo infinito. Revise la sección posterior "Ciclo Infinito".

- La mayoría de las veces, una recursión infinita causará que el programa se ejecute por un momento y luego produzca un error "RuntimeError: Maxi-

mum recursion depth exceeded”. Si esto ocurre revise la sección posterior “Recursión Infinita”.

Si no está obteniendo este error, pero sospecha que hay un problema con una función recursiva o método, también puede usar las técnicas de la sección “Recursión Infinita”.

- Si ninguno de estos pasos funciona, revise otros ciclos y otras funciones recursivas, o métodos.
- Si eso no funciona entonces es posible que usted no comprenda el flujo de ejecución que hay en su programa. Vaya a la sección posterior “Flujo de ejecución”.

Ciclo infinito

Si usted cree que tiene un ciclo infinito añada una sentencia `print` al final de éste que imprima los valores de las variables de ciclo (las que aparecen en la condición) y el valor de la condición.
Por ejemplo:

```
while x > 0 and y < 0 :
  # hace algo con x
  # hace algo con y

  print  "x: ", x
  print  "y: ", y
  print  "condicion: ", (x > 0 and y < 0)
```

Ahora, cuando ejecute el programa, usted verá tres líneas de salida para cada iteración del ciclo. En la última iteración la condición debe ser `falsa`. Si el ciclo sigue, usted podrá ver los valores de `x` y `y`, y puede deducir por qué no se están actualizando correctamente.

Recursión infinita

La mayoría de las veces una recursión infinita causará que el programa se ejecute durante un momento y luego producirá un error: `Maximum recursion depth exceeded`.
Si sospecha que una función o método está causando una recursión infinita, empiece por chequear la existencia de un caso base. En otras palabras, debe haber una condición que haga que el programa o método retorne sin hacer un llamado recursivo. Si no lo hay, es necesario reconsiderar el algoritmo e identificar un caso base.

Si hay un caso base, pero el programa no parece alcanzarlo, añada una sentencia `print` al principio de la función o método que imprima los parámetros. Ahora, cuando ejecute el programa usted verá unas pocas líneas de salida cada vez que la función o método es llamada, y podrá ver los parámetros. Si no están cambiando de valor acercándose al caso base, usted podrá deducir por qué ocurre esto.

Flujo de ejecución

Si no está seguro de cómo se mueve el flujo de ejecución a través de su programa, añada sentencias `print` al comienzo de cada función con un mensaje como "entrando a la función `foo`," donde `foo` es el nombre de la función.
Ahora, cuando ejecute el programa, se imprimirá una traza de cada función a medida que van siendo llamadas.

A.2.3. Cuando ejecuto el programa obtengo una excepción

Si algo va mal durante la ejecución, Python imprime un mensaje que incluye el nombre de la excepción, la línea del programa donde ocurrió, y un trazado inverso. El trazado inverso identifica la función que estaba ejecutándose, la función que la llamó, la función que llamó a *esta* última, y así sucesivamente. En otras palabras, traza el camino de llamados que lo llevaron al punto actual de ejecución. También incluye el número de línea en su archivo, donde cada uno de estos llamados tuvo lugar.
El primer paso consiste en examinar en el programa el lugar donde ocurrió el error y ver si se puede deducir qué pasó. Aquí están algunos de los errores en tiempo de ejecución más comunes:

NameError: usted está tratando de usar una variable que no existe en el ambiente actual. Recuerde que las variables locales son locales. No es posible referirse a ellas afuera de la función donde se definieron.

TypeError: hay varias causas:

- Usted está tratando de usar un valor impropiamente. Por ejemplo: indexar una cadena, lista o tupla con un valor que no es entero.
- No hay correspondencia entre los elementos en una cadena de formato y los elementos pasados para hacer la conversión. Esto puede pasar porque el número de elementos no coincide o porque se está pidiendo una conversión invalida.
- Usted está pasando el número incorrecto de argumentos a una función o método. Para los métodos, mire la definición de métodos y chequee que el primer parámetro sea `self`. Luego mire el llamado, asegúrese de

que se hace el llamado sobre un objeto del tipo correcto y de pasar los otros parámetros correctamente.

KeyError: usted está tratando de acceder a un elemento de un diccionario usando una llave que éste no contiene.

AttributeError: está tratando de acceder a un atributo o método que no existe.

IndexError: el índice que está usando para acceder a una lista, cadena o tupla es más grande que su longitud menos uno. Inmediatamente, antes de la línea del error, agregue una sentencia `print` para desplegar el valor del índice y la longitud del arreglo. ¿Tiene éste el tamaño correcto? ¿Tiene el índice el valor correcto?

A.2.4. Agregué tantas sentencias `print` que estoy inundado de texto de salida

Uno de los problemas de usar sentencias `print` para depurar es que uno puede terminar inundado de salida. Hay dos formas de proceder: simplificar la salida o simplificar el programa.

Para simplificar la salida se pueden eliminar o comentar sentencias `print` que no son de ayuda, o se pueden combinar, o se puede dar formato a la salida, de forma que quede más fácil de entender.

Para simplificar el programa hay varias cosas que se pueden hacer. Primero, disminuya la escala del problema que el programa intenta resolver. Por ejemplo, si usted está ordenando un arreglo, utilice uno *pequeño* como entrada. Si el programa toma entrada del usuario, pásele la entrada más simple que causa el problema.

Segundo, limpie el programa. Borre el código muerto y reorganízelo para hacerlo lo más legible que sea posible. Por ejemplo, si usted sospecha que el problema está en una sección de código profundamente anidada, intente reescribir esa parte con una estructura más sencilla. Si sospecha de una función grande, trate de partirla en funciones mas pequeñas y pruébelas separadamente.

Este proceso de encontrar el caso mínimo de prueba que activa el problema a menudo permite encontrar el error. Si usted encuentra que el programa funciona en una situación, pero no en otras, esto le da una pista de lo que está sucediendo.

Similarmente, reescribir un trozo de código puede ayudar a encontrar errores muy sutiles. Si usted hace un cambio que no debería alterar el comportamiento del programa, y sí lo hace, esto es una señal de alerta.

A.3. Errores semánticos

Estos son los mas difíciles de depurar porque ni el compilador ni el sistema de ejecución dan información sobre lo que está fallando. Sólo usted sabe lo que el programa debe hacer y sólo usted sabe por qué no lo está haciendo bien.

El primer paso consiste en hacer una conexión entre el código fuente del programa y el comportamiento que está dándose. Ahora, usted necesita una hipótesis sobre lo que el programa está haciendo realmente. Una de las cosas que complica el asunto es que la ejecución de programas en un computador moderno es muy rápida.

A veces deseará desacelerar el programa hasta una velocidad humana, y con algunos programas depuradores esto es posible. Pero el tiempo que toma insertar unas sentencias `print` bien situadas a menudo es mucho más corto comparado con la configuración del depurador, la inserción y eliminación de puntos de quiebre (breakpoints en inglés) y "saltar" por el programa al punto donde el error se da.

A.3.1. Mi programa no funciona

Usted debe hacerse estas preguntas:

- ¿Hay algo que el programa debería hacer, pero no hace? Encuentre la sección de código que tiene dicha funcionalidad y asegúrese de que se ejecuta en los momentos adecuados.
- ¿Está pasando algo que no debería? Encuentre código en su programa que tenga una funcionalidad y vea si ésta se ejecuta cuando no debería.
- ¿Hay una sección de código que produce un efecto que no esperaba usted? Asegúrese de que entiende dicha sección de código, especialmente si tiene llamados a funciones o métodos en otros módulos. Lea la documentación para las funciones que usted llama. Intente escribir casos de prueba más sencillos y chequee los resultados.

Para programar, usted necesita un modelo mental de cómo trabajan los programas. Si usted escribe un programa que no hace lo que se espera, muy frecuentemente el problema no está en el programa, sino en su modelo mental.

La mejor forma de corregir su modelo mental es descomponer el programa en sus componentes (usualmente funciones y métodos) para luego probarlos independientemente. Una vez encuentre la discrepancia entre su modelo y la realidad, el problema puede resolverse.

Por supuesto, usted debería construir y probar componentes a medida que desarrolla el programa. Si encuentra un problema, debería haber una pequeña cantidad de código nuevo que puede estar incorrecto.

A.3.2. He obtenido una expresión grande y peluda que no hace lo que espero

Escribir expresiones complejas está bien en tanto queden legibles, sin embargo, puede ser difícil depurarlas. Es una buena idea separar una expresión compleja en una serie de asignaciones a variables temporales.
Por ejemplo:

```
self.manos[i].agregarCarta(
  self.manos[self.encontrarVecino(i)].eliminarCarta())
```

Esto puede reescribirse como:

```
vecino = self.encontrarVecino (i)
cartaEscogida = self.manos[vecino].eliminarCarta()
self.manos[i].agregarCarta(cartaEscogida)
```

La versión explícita es más fácil de leer porque los nombres de variables proporcionan una documentación adicional y porque se pueden chequear los tipos de los valores intermedios desplegándolos.
Otro problema que ocurre con las expresiones grandes es que el orden de evaluación puede no ser el que usted espera. Por ejemplo, si usted está traduciendo la expresión $\frac{x}{2\pi}$ a Python, podría escribir:

```
y = x / 2 * math.pi;
```

Esto es incorrecto, porque la multiplicación y la división tienen la misma precedencia y se evalúan de izquierda a derecha. Así que ese código calcula $x\pi/2$.
Una buena forma de depurar expresiones es agregar paréntesis para hacer explícito el orden de evaluación:

```
 y = x / (2 * math.pi);
```

Cuando no esté seguro del orden de evaluación, use paréntesis. No sólo corregirá el programa si había un error, sino que también lo hará mas legible para otras personas que no se sepan las reglas de precedencia.

A.3.3. Tengo una función o método que no retorna lo que debería

Si usted tiene una sentencia `return` con una expresión compleja no hay posibilidad de imprimir el valor del `return` antes de retornar. Aquí también se puede usar una variable temporal. Por ejemplo, en vez de:

```
return self.manos[i].eliminarParejas()
```

se podría escribir:

```
cont = self.manos[i].eliminarParejas()
return cont
```

Ahora usted tiene la oportunidad de desplegar el valor de `count` antes de retornar.

A.3.4. Estoy REALMENTE atascado y necesito ayuda

Primero, intente alejarse del computador por unos minutos. Los computadores emiten ondas que afectan al cerebro causando estos efectos:

- Frustración e ira.
- Creencias supersticiosas ("el computador me odia") y pensamiento mágico ("el programa sólo funciona cuando me pongo la gorra al revés").
- Programación aleatoria (el intento de programar escribiendo cualquier programa posible y escogiendo posteriormente el que funcione correctamente).

Si usted está sufriendo de alguno de estos síntomas, tómese un paseo. Cuando ya esté calmado, piense en el programa. ¿Qué está haciendo? ¿Cuáles son las causas posibles de éste comportamiento? ¿Cuándo fue la última vez que funcionaba bien, y qué hizo usted después de eso?
Algunas veces, sólo toma un poco de tiempo encontrar un error. A menudo encontramos errores cuando estamos lejos del computador y dejamos que la mente divague. Algunos de los mejores lugares para encontrar errores son los trenes, duchas y la cama, justo antes de dormir.

A.3.5. No, realmente necesito ayuda

Esto sucede. Hasta los mejores programadores se atascan alguna vez. A veces se ha trabajado tanto tiempo en un programa que ya no se puede ver el error. Un par de ojos frescos es lo que se necesita.
Antes de acudir a alguien más, asegúrese de agotar todas las técnicas descritas aquí. Su programa debe ser tan sencillo como sea posible y usted debería encontrar la entrada más pequeña que causa el error. Debería tener sentencias `print` en lugares apropiados (y la salida que despliegan debe ser comprensible). Usted debe entender el problema lo suficientemente bien como para describirlo concisamente.
Cuando acuda a alguien, asegúrese de darle la información necesaria:

- Si hay un mensaje de error, ¿cuál es, y a qué parte del programa se refiere?

- ¿Cuál fue el último cambio antes de que se presentara el error? ¿Cuáles fueron las últimas líneas de código que escribió usted o cuál es el nuevo caso de prueba que falla?
- ¿Qué ha intentado hasta ahora, y qué ha aprendido sobre el programa?

Cuando encuentre el error, tómese un segundo para pensar sobre lo que podría haber realizado para encontrarlo más rápido. La próxima vez que le ocurra algo similar, será capaz de encontrar el error rápidamente.
Recuerde, el objetivo no sólo es hacer que el programa funcione. El objetivo es aprender cómo hacer que los programas funcionen.

Apéndice B

Creando un nuevo tipo de datos

Los lenguajes de programación orientados a objetos permiten a los programadores crear nuevos tipos de datos que se comportan de manera muy similar a los tipos primitivos. Exploraremos esta característica construyendo una clase `Fraccionario` que se comporte como los tipos de datos numéricos primitivos (enteros y flotantes).

Los números fraccionario o racionales son valores que se pueden expresar como una división entre dos números enteros, como $\frac{1}{3}$. El número superior es el numerador y el inferior es es el denominador.

La clase `Fraccion` empieza con un método constructor que recibe como parámetros al numerador y al denominador:

```
class Fraccion:
  def __init__(self, numerador, denominador=1):
    self.numerador = numerador
    self.denominador = denominador
```

El denominador es opcional. Una Fracción con un solo parámetro representa a un número entero. Si el numerador es n, construimos la fracción $n/1$.

El siguiente paso consiste en escribir un método `__str__` que despliegue las fracciones de una manera natural. Como estamos acostumbrados a la notación "numerador/denominador", lo más natural es:

```
class Fraccion:
  ...
  def __str__(self):
    return "%d/%d" % (self.numerador, self.denominador)
```

Para realizar pruebas, ponemos este código en un archivo `Fraccion.py` y lo importamos en el intérprete de Python. Ahora creamos un objeto fracción y lo imprimimos.

```
>>> from Fraccion import Fraccion
>>> s = Fraccion(5,6)
>>> print "La fraccion es", s
La fraccion es 5/6
```

El método `print`, automáticamente invoca al método `__str__` de manera implícita.

B.1. Multiplicación de fracciones

Nos gustaría aplicar los mismos operadores de suma, resta, multiplicación y división a las fracciones. Para lograr esto podemos sobrecargar los operadores matemáticos en la clase `Fraccion`.
La multiplicación es la operación más sencilla entre fraccionarios. El resultado de multiplicar dos fracciones (a y b) es una nueva fracción en la que el numerador es el producto de los dos numeradores y el denominador es el producto de los dos denominadores.
Python permite que el método `__mul__` se pueda definir en una clase para sobrecargar el operador `*`:

```
class Fraccion:
  ...
  def __mul__(self, otro):
    return Fraccion(self.numerador*otro.numerador,
                    self.denominador*otro.denominador)
```

Podemos probar este método calculando un producto sencillo:

```
>>> print Fraccion(5,6) * Fraccion(3,4)
15/24
```

Funciona, pero se puede mejorar. Podemos manejar el caso en el que se multiplique una fracción por un número entero. Por medio de la función `type` se puede probar si `otro` es un entero y convertirlo a una fracción antes de realizar el producto:

```
class Fraccion:
  ...
  def __mul__(self, otro):
    if type(otro) == type(5):
      otro = Fraccion(otro)
    return Fraccion(self.numerador   * otro.numerador,
                    self.denominador * otro.denominador)
```

Ahora, la multiplicación entre enteros y fracciones funciona, pero sólo si la fracción es el operando a la izquierda :

```
>>> print Fraccion(5,6) * 4
20/6
>>> print 4 * Fraccion(5,6)
TypeError: __mul__ nor __rmul__ defined for these operands
```

Para evaluar un operador binario como la multiplicación, Python chequea el operando izquierdo primero, para ver si su clase define el método `__mul__`, y que tenga soporte para el tipo del segundo operando. En este caso el operador primitivo para multiplicar enteros no soporta las fracciones.
Después, Python chequea si el operando a la derecha provee un método `__rmul__` que soporte el tipo del operando de la izquierda. En este caso, como no hay definición de `__rmul__` en la clase `Fraccion`, se genera un error de tipo.
Hay una forma sencilla de definir `__rmul__`:

```
class Fraccion:
  ...
  __rmul__ = __mul__
```

Esta asignación dice que `__rmul__` contiene el mismo código que `__mul__`. Si ahora evaluamos `4 * Fraccion(5,6)`, Python llama a `__rmul__` y le pasa al 4 como parámetro:

```
>>> print 4 * Fraccion(5,6)
20/6
```

Como `__rmul__` tiene el mismo código que `__mul__`, y el método `__mul__` puede recibir un parámetro entero, nuestra multiplicación de fracciones funciona bien.

B.2. Suma de fracciones

La suma es más complicada que la multiplicación. La suma $a/b + c/d$ da como resultado $\frac{(ad+cb)}{bd}$.
Basándonos en la multiplicación, podemos escribir los métodos `__add__` y `__radd__`:

```
class Fraccion:
  ...
  def __add__(self, otro):
    if type(otro) == type(5):
      otro = Fraccion(otro)
    return Fraccion(self.numerador   * otro.denominador +
```

```
                    self.denominador * otro.numerador,
                    self.denominador * otro.denominador)

  __radd__ = __add__
```

Podemos probar estos métodos con objetos `Fraccion` y con números enteros.

```
>>> print Fraccion(5,6) + Fraccion(5,6)
60/36
>>> print Fraccion(5,6) + 3
23/6
>>> print 2 + Fraccion(5,6)
17/6
```

Los primeros ejemplos llaman al método `__add__`; el último ejemplo llama al método `__radd__`.

B.3. El algoritmo de Euclides

En el ejemplo anterior, calculamos $5/6 + 5/6$ y obtuvimos $60/36$. Es correcto, pero no es la manera más sencilla de presentar la respuesta. Para **simplificar** la fracción tenemos que dividir el numerador y el denominador por su **máximo divisor común (MDC)**, que para este caso es 12. Entonces, un resultado mas sencillo es $5/3$.

En general, cada vez que creamos un nuevo objeto de tipo `Fraccion` deberiamos simplificarlo dividiendo el numerador y el denominador por su MDC. Si la fracción no se puede simplificar, el MDC es 1.

Euclides de Alejandría (aprox. 325–265 A.C) presentó un algoritmo para encontrar el MDC de dos números enteros m y n:

> Si n divide a m exactamente, entonces n es el MDC. Sino, el MDC de m y n es el MDC de n y el residuo de la división m/n.

Esta definición recursiva se puede implementar en una función:

```
def MDC (m, n):
  if m % n == 0:
    return n
  else:
    return MDC(n, m%n)
```

En la primera línea el operador residuo nos permite chequear si n divide a n exactamente. En la última línea, lo usamos para calcular el residuo de la división.

Como todas las operaciones que hemos escrito crean nuevas fracciones como resultado, podemos simplificar todos los valores de retorno modificando el método constructor.

```
class Fraccion:
  def __init__(self, numerador, denominador=1):
    g = MDC (numerador, denominador)
    self.numerador   =   numerador / g
    self.denominador = denominador / g
```

Ahora, cada vez que creamos una nueva `Fraccion`, ¡se simplifica!.

```
>>> Fraccion(100,-36)
-25/9
```

Una característica adicional que nos provee `MDC` es que si la fracción es negativa, el signo menos siempre se mueve hacia el numerador.

B.4. Comparando fracciones

Si vamos a comparar dos objetos `Fraccion`, digamos `a` y `b`, evaluando la expresión `a == b`, como la implementación de `==` chequea igualdad superficial de objetos por defecto, sólo retornará cierto si `a` y `b` *son* el mismo objeto.
Es mucho más probable que deseemos retornar cierto si a y b tienen el mismo valor —esto es, chequear igualdad profunda.
Tenemos que enseñarle a las fracciones cómo compararse entre sí. Como vimos en la sección 16.4, podemos sobrecargar todos los operadores de comparación por medio de la implementación de un método `__cmp__`.
Por convención, el método `__cmp__` retorna un número negativo si `self` es menos que `otro`, cero si son iguales, y un número positivo si `self` es mayor que `otro`.
La forma más sencilla de comparar fracciones consiste en hacer una multiplicación cruzada. Si $a/b > c/d$, entonces $ad > bc$. Con esto en mente, implementamos `__cmp__`:

```
class Fraccion:
  ...
  def __cmp__(self, otro):
    dif = (self.numerador  * otro.denominador -
           otro.numerador * self.denominador)
    return dif
```

Si `self` es mayor que `otro`, entonces `dif` será positivo. Si `otro` es mayor, `dif` será negativo. Si son iguales, `dif` es cero.

B.5. Extendiendo las fracciones

Todavía no hemos terminado. Tenemos que implementar la resta sobrecargando el método `__sub__` y la división con el método `__div__`.
Podemos restar por medio de la suma si antes negamos (cambiamos de signo) al segundo operando. También podemos dividir por medio de la multiplicación si antes invertimos el segundo operando.
Siguiendo este razonamiento, una forma de realizar las operaciones resta y división consiste en definir primero la negación por medio de la sobrecarga de `__neg__` y la inversión sobre sobrecargando a `__invert__`.
Un paso adicional sería implementar `__rsub__` y `__rdiv__`. Desafortunadamente no podemos usar el mismo truco que aplicamos para la suma y la multiplicación, porque la resta y la división no son conmutativas. En estas operaciones el orden de los operandos altera el resultado, así que no podemos asignar a `__rsub__` y a `__rdiv__` los método `__sub__` y `__div__`, respectivamente.
Para realizar la **negación unaria**, sobrecargamos a `__neg__`. Podemos calcular potencias sobrecargando a `__pow__`, pero la implementación tiene un caso difícil: si el exponente no es un entero, puede que no sea posible representar el resultado como una `Fraccion`. Por ejemplo, la siguiente expresión `Fraccion(2) ** Fraccion(1,2)` es la raíz cuadrada de 2, que es un número irracional (no puede representarse por ninguna fracción). Así que no es fácil escribir una función general para `__pow__`.
Hay otra extensión a la clase `Fraccion` que usted puede imaginar. Hasta aquí, hemos asumido que el numerador y el denominador son enteros. También podemos permitir que sean de tipo long.

> *Como ejercicio, complemente la implementación de la clase* `Fraccion` *para que permita las operaciones de resta, división y exponenciación. Además, debe soportar denominadores y numeradores de tipo long (enteros grandes).*

B.6. Glosario

Máximo divisor común (MDC): el entero positivo más grande que divide exactamente a dos números (por ejemplo, el numerador y el denominador en una fracción).

Simplificar: cambiar una fracción en otra equivalente que tenga un MDC de 1.

negación unaria: la operación que calcula un inverso aditivo, usualmente representada con un signo menos. Es "unaria" en contraposición con el menos binario que representa a la resta.

Apéndice C

Programas completos

C.1. Clase punto

```
class Punto:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ',' + str(self.y) + ')'

  def __add__(self, otro):
    return Punto(self.x + otro.x, self.y + otro.y)

  def __sub__(self, otro):
    return Punto(self.x - otro.x, self.y - otro.y)

  def __mul__(self, otro):
    return self.x * otro.x + self.y * otro.y

  def __rmul__(self, otro):
    return Punto(otro * self.x, otro * self.y)

  def invertir(self):
    self.x, self.y = self.y, self.x

  def DerechoYAlReves(derecho):
    from copy import copy
```

```
    alreves = copy(derecho)
    alreves.invertir()
    print str(derecho) + str(alreves)
```

C.2. Clase hora

```
class Hora:
  def __init__(self, hora=0, minutos=0, segundos=0):
    self.hora = hora
    self.minutos = minutos
    self.segundos = segundos

  def __str__(self):
    return str(self.hora) + ":" + str(self.minutos) +
           ":" + str(self.segundos)

  def convertirAsegundos(self):
    minutos = self.hora * 60 + self.minutos
    segundos = self.minutos * 60 + self.segundos
    return segundos

  def incrementar(self, segs):
    segs = segs + self.segundos

    self.hora = self.hora + segs/3600
    segs = segs % 3600
    self.minutos = self.minutos + segs/60
    segs = segs % 60
    self.segundos = segs

def crearHora(segs):
  H = Hora()
  H.hora = segs/3600
  segs = segs - H.hora * 3600
  H.minutos = segs/60
  segs = segs - H.minutos * 60
  H.segundos = segs
  return H
```

C.3. Cartas, mazos y juegos

```
import random

class Carta:
  listaFiguras = ["Treboles", "Diamantes",
                  "Corazones", "Picas"]
  listaValores = [ "narf", "As", "2", "3", "4", "5", "6",
               "7","8", "9", "10","Jota", "Reina", "Rey"]

  def __init__(self, figura=0, valor=0):
    self.figura = figura
    self.valor = valor

  def __str__(self):
    return self.listaValores[self.valor] + " de "
           + self.listaFiguras[self.figura]

  def __cmp__(self, otro):
    # revisa las figuras
    if self.figura > otro.figura:
       return 1
    if self.figura < otro.figura:
       return -1
    # las figuras son iguales, se chequean los valores
    if self.valor > otro.valor:
      return 1
    if self.valor < otro.valor:
      return -1
    # los valores son iguales,  hay empate
    return 0

class Mazo:
  def __init__(self):
    self.Cartas = []
    for figura in range(4):
      for valor in range(1, 14):
        self.Cartas.append(Carta(figura, valor))

  def imprimirMazo(self):
    for Carta in self.Cartas:
      print Carta
```

```
  def __str__(self):
    s = ""
    for i in range(len(self.Cartas)):
      s = s + " "*i + str(self.Cartas[i]) + "\n"
    return s

  def barajar(self):
    import random
    nCartas = len(self.Cartas)
    for i in range(nCartas):
      j = random.randrange(i, nCartas)
      [self.Cartas[i], self.Cartas[j]] = [self.Cartas[j],
                                          self.Cartas[i]]

  def eliminarCarta(self, Carta):
    if Carta in self.Cartas:
      self.Cartas.remove(Carta)
      return 1
    else: return 0

  def entregarCarta(self):
    return self.Cartas.pop()

  def estaVacio(self):
    return (len(self.Cartas) == 0)

  def repartir(self, manos, nCartas=999):
    nmanos = len(manos)
    for i in range(nCartas):
      if self.estaVacio():
         break    # rompe el ciclo si no hay cartas
      # quita la carta del tope
      Carta = self.entregarCarta()
      # quien tiene el proximo turnoo?
      mano = manos[i % nmanos]
      # agrega la carta a la mano
      mano.agregarCarta(Carta)

class mano(Mazo):
  def __init__(self, nombre=""):
    self.Cartas = []
```

```
    self.nombre = nombre

  def agregarCarta(self,Carta) :
    self.Cartas.append(Carta)

  def __str__(self):
    s = "mano " + self.nombre
    if self.estaVacio():
      s = s + " esta vacia\n"
    else:
      s = s + " contiene\n"
    return s + Mazo.__str__(self)

class JuegoCartas:
  def __init__(self):
    self.Mazo = Mazo()
    self.Mazo.barajar()

class ManoJuegoSolterona(mano):
  def eliminarParejas(self):
    cont = 0
    originalCartas = self.Cartas[:]
    for carta in originalCartas:
      m = Carta(3-carta.figura, carta.valor)
      if m in self.Cartas:
        self.Cartas.remove(carta)
        self.Cartas.remove(m)
        print "mano %s: %s parejas %s" %
                 (self.nombre,carta,m)
        cont = cont+1
    return cont

class JuegoSolterona(JuegoCartas):
  def jugar(self, nombres):
    # elimina la reina de treboles
    self.Mazo.eliminarCarta(Carta(0,12))

    # crea manos con base en los nombres
    self.manos = []
    for nombre in nombres :
        self.manos.append(ManoJuegoSolterona(nombre))
```

```
    # reparte las Cartas
    self.Mazo.repartir(self.manos)
    print "---------- Cartas se han repartido"
    self.imprimirmanos()

    # eliminar parejas iniciales
    parejas = self.eliminarParejas()
    print "----- parejas descartadas, empieza el juego"
    self.imprimirmanos()

    # jugar hasta que se eliminan 50 cartas
    turno = 0
    nummanos = len(self.manos)
    while parejas < 25:
      parejas = parejas + self.jugarUnturno(turno)
      turno = (turno + 1) % nummanos

    print "---------- Juego Terminado"
    self.imprimirmanos ()

  def eliminarParejas(self):
    cont = 0
    for mano in self.manos:
      cont = cont + mano.eliminarParejas()
    return cont

  def jugarUnturno(self, i):
    if self.manos[i].estaVacio():
      return 0
    vecino = self.encontrarvecino(i)
    cartaEscogida = self.manos[vecino].entregarCarta()
    self.manos[i].agregarCarta(cartaEscogida)
    print "mano", self.manos[i].nombre,
          "escogió", cartaEscogida
    count = self.manos[i].eliminarParejas()
    self.manos[i].barajar()
    return count

  def encontrarvecino(self, i):
    nummanos = len(self.manos)
    for siguiente in range(1,nummanos):
      vecino = (i + siguiente) % nummanos
```

```
      if not self.manos[vecino].estaVacio():
        return vecino

  def imprimirmanos(self) :
    for mano in self.manos :
      print mano
```

C.4. Listas enlazadas

```
def imprimirlista(Nodo) :
  while Nodo :
    print Nodo,
    Nodo = Nodo.siguiente
  print

def imprimirAlReves(lista) :
  if lista == None :
      return
  cabeza = lista
  resto = lista.siguiente
  imprimirAlReves(resto)
  print cabeza,

def imprimirAlRevesBien(lista) :
  print "(",
  if lista != None :
    cabeza = lista
    resto = lista.siguiente
    imprimirAlReves(resto)
    print cabeza,
  print ")",

def eliminarSegundo(lista) :
  if lista == None : return
  first  = lista
  second = lista.siguiente
  first.siguiente = second.siguiente
  second.siguiente = None
  return second
```

```
class Nodo :
  def __init__(self, carga=None) :
    self.carga = carga
    self.siguiente  = None

  def __str__(self) :
    return str(self.carga)

  def imprimirAlReves(self) :
    if self.siguiente != None :
      resto = self.siguiente
      resto.imprimirAlReves()
    print self.carga,

class ListaEnlazada :
  def __init__(self) :
    self.numElementos = 0
    self.cabeza   = None

  def imprimirAlReves(self) :
    print "(",
    if self.cabeza != None :
      self.cabeza.imprimirAlReves()
    print ")",

  def agregarAlPrincipio(self, carga) :
    nodo = Nodo(carga)
    nodo.siguiente = self.cabeza
    self.cabeza = nodo
    self.numElementos = self.numElementos + 1
```

C.5. Clase pila

```
class Pila:
  def __init__(self):
    self.items = []

  def meter(self, item):
    self.items.append(item)

  def sacar(self):
```

```
    return self.items.pop()

  def estaVacia(self):
    return(self.items == [])

def evalPostfija(expr) :
  import re
  expr = re.split("([^0-9])", expr)
  pila = Pila()
  for lexema in expr :
    if  lexema == '' or lexema == ' ':
      continue
    if  lexema == '+' :
      suma = pila.sacar() + pila.sacar()
      pila.meter(suma)
    elif lexema == '*' :
      producto = pila.sacar() * pila.sacar()
      pila.meter(producto)
    else :
      pila.meter(int(lexema))
  return pila.sacar()
```

C.6. Colas PEPS y de colas de prioridad

```
class Cola:
  def __init__(self):
    self.numElementos = 0
    self.primero = None

  def estaVacia(self):
    return (self.numElementos == 0)

  def meter(self, carga):
    nodo = Nodo(carga)
    nodo.siguiente = None
    if self.primero == None:
       # si esta vacia este nodo sera el primero
       self.primero = nodo
    else:
       # encontrar el ultimo nodo
```

```
       ultimo = self.primero
       while ultimo.siguiente:
 ultimo = ultimo.siguiente
 # pegar el nuevo
 ultimo.siguiente = nodo
       self.numElementos = self.numElementos + 1

  def sacar(self):
    carga = self.primero.carga
    self.primero = self.primero.siguiente
    self.numElementos = self.numElementos - 1
    return carga

class ColaMejorada:
  def __init__(self):
    self.numElementos = 0
    self.primero = None
    self.ultimo = None

  def estaVacia(self):
    return (self.numElementos == 0)

  def meter(self, carga):
    nodo = Nodo(carga)
    nodo.siguiente = None
    if self.numElementos == 0:
      # si esta vacia, el nuevo nodo
      # es primero y ultimo
      self.primero = self.ultimo = nodo
    else:
      # encontrar el ultimo nodo
      ultimo = self.ultimo
      # pegar el nuevo nodo
      ultimo.siguiente = nodo
      self.ultimo = nodo
      self.numElementos = self.numElementos + 1

    def sacar(self):
      carga = self.primero.carga
      self.primero = self.primero.siguiente
      self.numElementos = self.numElementos - 1
      if self.numElementos == 0:
```

```
        self.ultimo = None
      return carga

class ColaPrioridad:
  def __init__(self):
    self.items = []

  def estaVacia(self):
    return self.items == []

  def meter(self, item):
    self.items.append(item)

  def eliminar(self) :
    maxi = 0
    for i in range(1,len(self.items)) :
      if self.items[i] > self.items[maxi] :
maxi = i
    item = self.items[maxi]
    self.items[maxi:maxi+1] = []
    return item

  def sacar(self):
    maxi = 0
    for i in range(0,len(self.items)):
      if self.items[i] > self.items[maxi]:
maxi = i
      item = self.items[maxi]
      self.items[maxi:maxi+1] = []
      return item


class golfista:
  def __init__(self, nombre, puntaje):
    self.nombre = nombre
    self.puntaje= puntaje

  def __str__(self):
    return "%-16s: %d" % (self.nombre, self.puntaje)

  def __cmp__(self, otro):
    # el menor tiene mayor prioridad
```

```
    if self.puntaje < otro.puntaje: return 1
      if self.puntaje > otro.puntaje: return -1
        return 0
```

C.7. Árboles

```
class Arbol:
  def __init__(self, carga, izquierdo=None, derecho=None):
    self.carga = carga
    self.izquierdo = izquierdo
    self.derecho = derecho

  def __str__(self):
    return str(self.carga)

  def obtenerizquierdo(self):
    return self.izquierdo

  def obtenerderecho(self):
    return self.derecho

  def obtenercarga(self):
    return self.carga

  def asignarcarga(self, carga):
    self.carga = carga

  def asignarizquierdo(self, i):
    self.izquierdo = i

  def asignarderecho(self, d):
    self.derecho = d

def total(arbol):
  if arbol.izquierdo == None or arbol.derecho== None:
    return arbol.carga
  else:
    return total(arbol.izquierdo) + total(arbol.derecho) +
             arbol.carga
```

```
def imprimirarbol(arbol):
  if arbol == None:
    return
  else:
    print arbol.carga
    imprimirarbol(arbol.izquierdo)
    imprimirarbol(arbol.derecho)

def imprimirarbolPostorden(arbol):
  if arbol == None:
    return
  else:
    imprimirarbolPostorden(arbol.izquierdo)
    imprimirarbolPostorden(arbol.derecho)
    print arbol.carga

def imprimirabolEnOrden(arbol):
  if arbol == None:
    return
  imprimirabolEnOrden(arbol.izquierdo)
  print arbol.carga,imprimirabolEnOrden(arbol.derecho)

def imprimirarbolSangrado(arbol, nivel=0):
  if arbol == None:
    return
  imprimirarbolSangrado(arbol.derecho, nivel+1)
  print " "*nivel + str(arbol.carga)
  imprimirarbolSangrado(arbol.izquierdo, nivel+1)
```

C.8. Árboles de expresiones

```
def obtenerLexema(listaLexemas, esperado):
  if listaLexemas[0] == esperado:
      del listaLexemas[0]
      return 1
  else:
      return 0

def obtenerNumero(listaLexemas):
  x = listaLexemas[0]
  if type(x) != type(0):
```

```
      return None
  del listaLexemas[0]
  return arbol (x, None, None)


def obtenerProducto(listaLexemas):
  a = obtenerNumero(listaLexemas)
  if obtenerLexema(listaLexemas, "*"):
    b = obtenerProducto(listaLexemas)
    return arbol ("*", a, b)
  else:
    return a

def obtenerNumero(listaLexemas):
  if obtenerLexema(listaLexemas, "("):
    # obtiene la subexpresión
    x = obtenerSuma(listaLexemas)
    # elimina los paréntesis
    obtenerLexema(listaLexemas, ")")
    return x
  else:
    x = listaLexemas[0]
    if type(x) != type(0):
      return None
    listaLexemas[0:1] = []
   return Arbol (x, None, None)
```

C.9. Adivinar el animal

```
def animal():
  # Un solo nodo
  raiz = Arbol("pajaro")
      # Hasta que el usuario salga
  while True:
    print
    if not si("Esta pensando en un animal? "):
break
    # Recorrer el arbol
    arbol = raiz
    while arbol.obtenerizquierdo() != None:
```

```
      pregunta = arbol.obtenercarga() + "? "
      if si(pregunta):
  arbol = arbol.obtenerderecho()
      else:
  arbol = arbol.obtenerizquierdo()
    # conjetura
    conjetura = arbol.obtenercarga()
    pregunta = "¿Es un" + conjetura + "? "
    if si(pregunta):
      print "¡Soy el mejor!"
      continue
    # obtener mas informacion
    pregunta = "¿Cual es el nombre el animal? "
    animal = raw_input(pregunta)
    pregunta = "¿Que pregunta permitiria distinguir
un %s de un %s? "
    q = raw_input(pregunta % (animal,conjetura))
    # agrega un nuevo nodo arbol
    arbol.asignarcarga(q)
    pregunta = "¿Si el animal fuera %s
la respuesta sería? "
    if si(pregunta % animal):
      arbol.asignarizquierdo(Arbol(conjetura))
      arbol.asignarderecho(Arbol(animal))
    else:
      arbol.asignarizquierdo(Arbol(animal))
      arbol.asignarderecho(Arbol(conjetura))

def si(preg):
  from string import lower
  r = lower(raw_input(preg))
  return (r[0] == 's')
```

C.10. Clase Fraccion

```
class Fraccion:
  def __init__(self,numerador,denominador=1):
    self.numerador = numerador
    self.denominador = denominador
```

```
  def __str__(self):
    return "%d/%d" % (self.numerador, self.denominador)

  def __mul__(self, otro):
    if type(otro) == type(5):
      otro = Fraccion(otro)
    return Fraccion(self.numerador * otro.numerador,
    self.denominador * otro.denominador)

  __rmul__ = __mul__

  #suma de fracciones
  def __add__(self, otro):
    if type(otro) == type(5):
      otro = Fraccion(otro)
    return Fraccion(self.numerador * otro.denominador +
    self.denominador * otro.numerador,
    self.denominador * otro.denominador)

  __radd__ = __add__

  def __init__(self, numerador, denominador=1):
    g=MDC(numerador,denominador)
    self.numerador=numerador / g
    self.denominador=denominador / g

  def __cmp__(self, otro):
    dif = (self.numerador * otro.denominador -
   otro.numerador * self.denominador)
    return dif

def MDC (m,n):
      if m%n==0:
  return n
      else:
  return MDC(n,m%n)
```

Apéndice D

Lecturas adicionales recomendadas

¿Así que, hacia adónde ir desde aquí? Hay muchas direcciones para avanzar, extender su conocimiento de Python específicamente y sobre la ciencia de la computación en general.
Los ejemplos en este libro han sido deliberadamente sencillos, por esto no han mostrado las capacidades más excitantes de Python. Aquí hay una pequeña muestra de las extensiones de Python y sugerencias de proyectos que las utilizan.

- La programación de interfaces gráficas de usario, GUI (graphical user interfaces) permite a sus programas usar un sistema de ventanas para interactuar con el usuario y desplegar gráficos.

  El paquete gráfico para Python más viejo es Tkinter, que se basa en los lenguajes de guión Tcl y Tk de Jon Ousterhout. Tkinter viene empacado con la distribución Python.

  Otra plataforma de desarrollo popular es wxPython, que es una simple capa Python sobre wxWindows, un paquete escrito en C++ que implementa ventanas usando interfaces nativas en plataformas Windows y Unix (incluyendo a Linux). Las ventanas y controles en wxPython tienden a tener un aspecto más nativo que las que se realizan con Tkinter; también es más sencillo desarrollarlas.

  Cualquier tipo de programación de interfaces gráficas lo conducirá a la programación dirigida por eventos, donde el usuario y no el programador determinan el flujo de ejecución. Este estilo de programación requiere un tiempo para acostumbrarse y algunas veces implica reconsiderar la estructura de los programas.

- La programación en la Web integra a Python con Internet. Por ejemplo, usted puede construir programas cliente que abran y lean páginas remotas (casi) tan fácilmente como abren un archivo en disco. También hay módulos en Python que permiten acceder a archivos remotos por medio de ftp, y otros que posibilitan enviar y recibir correo electrónico. Python también se usa extensivamente para desarrollar servidores web que presten servicios.
- Las bases de datos son como superarchivos, donde la información se almacena en esquemas predefinidos y las relaciones entre los datos permiten navegar por ellos de diferentes formas. Python tiene varios módulos que facilitan conectar programas a varios motores de bases de datos, de código abierto y comerciales.
- La programación con hilos permite ejecutar diferentes hilos de ejecución dentro de un mismo programa. Si usted ha tenido la experiencia de desplazarse al inicio de una página web mientras el navegador continúa cargando el resto de ella, entonces tiene una noción de lo que los hilos pueden lograr.
- Cuando la preocupación es la velocidad, se pueden escribir extensiones a Python en un lenguaje compilado como C o C++. Estas extensiones forman la base de muchos módulos en la biblioteca de Python. El mecanismo para enlazar funciones y datos es algo complejo. La herramienta SWIG (Simplified Wrapper and Interface Generator) simplifica mucho estas tareas.

D.1. Libros y sitios web relacionados con Python

Aquí están las recomendaciones de los autores sobre sitios web:

- La página web de `www.python.org` es el lugar para empezar cualquier búsqueda de material relacionado con Python. Encontrará ayuda, documentación, enlaces a otros sitios, SIG (Special Interest Groups), y listas de correo a las que se puede unir.
- EL proyecto Open Book `www.ibiblio.com/obp` no sólo contiene este libro en línea, también los libros similares para Java y C++ de Allen Downey. Además, está *Lessons in Electric Circuits* de Tony R. Kuphaldt, *Getting down with ...* un conjunto de tutoriales (que cubren varios tópicos en ciencias de la computación) que han sido escritos y editados por estudiantes universitarios; *Python for Fun*, un conjunto de casos de estudio en Python escrito por Chris Meyers, y *The Linux Cookbook* de Michael Stultz, con 300 páginas de consejos y sugerencias.
- Finalmente si usted Googlea la cadena "python -snake -monty" obtendrá unos 337000000 resultados.

Aquí hay algunos libros que contienen más material sobre el lenguaje Python:

- *Core Python Programming*, de Wesley Chun, es un gran libro de 750 páginas, aproximadamente. La primera parte cubre las características básicas. La segunda introduce adecuadamente muchos tópicos más avanzados, incluyendo muchos de los que mencionamos anteriormente.
- *Python Essential Reference*, de David M. Beazley, es un pequeño libro, pero contiene mucha información sobre el lenguaje y la biblioteca estándar. También provee un excelente índice.
- *Python Pocket Reference*, de Mark Lutz, realmente cabe en su bolsillo. Aunque no es tan comprensivo como *Python Essential Reference*; es una buena referencia para las funciones y módulos más usados. Mark Lutz también es el autor de *Programming Python*, uno de los primeros (y más grandes) libros sobre Python que no está dirigido a novatos. Su libro posterior *Learning Python* es más pequeño y accesible.
- *Python Programming on Win32*, de Mark Hammond y Andy Robinson, se "debe tener" si pretende construir aplicaciones para el sistema operativo Windows. Entre otras cosas cubre la integración entre Python y COM, construye una pequeña aplicación con wxPython, e incluso realiza guiones que agregan funcionalidad a aplicaciones como Word y Excel.

D.2. Libros generales de ciencias de la computación recomendados

Las siguientes sugerencias de lectura incluyen muchos de los libros favoritos de los autores. Tratan sobre buenas prácticas de programación y las ciencias de la computación en general.

- *The Practice of Programming* de Kernighan y Pike no sólo cubre el diseño y la codificación de algoritmos y estructuras de datos, sino que también trata la depuración, las pruebas y la optimización de los programas. La mayoría de los ejemplos está escrita en C++ y Java, no hay ninguno en Python.
- *The Elements of Java Style*, editado por Al Vermeulen, es otro libro pequeño que discute algunos de los puntos mas sutiles de la buena programación, como el uso de buenas convenciones para los nombres, comentarios, incluso el uso de los espacios en blanco y la indentación (algo que no es problema en Python). El libro también cubre la programación por contrato que usa aserciones para atrapar errores mediante el chequeo de pre y postcondiciones, y la programación multihilo.

- *Programming Pearls*, de Jon Bentley, es un libro clásico. Comprende varios casos de estudio que aparecieron originalmente en la columna del autor en las *Communications of the ACM*. Los estudios examinan los compromisos que hay que tomar cuando se programa y por qué tan a menudo es una mala idea apresurarse con la primera idea que se tiene para desarrollar un programa. Este libro es uno poco más viejo que los otros (1986), así que los ejemplos están escritos en lenguajes más viejos. Hay muchos problemas para resolver, algunos traen pistas y otros su solución. Este libro fue muy popular, incluso hay un segundo volumen.

- *The New Turing Omnibus*, de A.K Dewdney, hace una amable introducción a 66 tópicos en ciencias de la computación, que van desde la computación paralela hasta los virus de computador, desde escanografías hasta algoritmos genéticos. Todos son cortos e interesantes. Un libro anterior de Dewdney *The Armchair Universe* es una colección de artículos de su columna *Computer Recreations* en la revista *Scientific American (Investigación y Ciencia)*, estos libros son una rica fuente de ideas para emprender proyectos.

- *Turtles, Termites and Traffic Jams*, de Mitchel Resnick, trata sobre el poder de la descentralización y cómo el comportamiento complejo puede emerger de la simple actividad coordinada de una multitud de agentes. Introduce el lenguaje StarLogo que permite escribir programas multiagentes. Los programas examinados demuestran el comportamiento complejo agregado, que a menudo es contraintuitivo. Muchos de estos programas fueron escritos por estudiantes de colegio y universidad. Programas similares pueden escribirse en Python usando hilos y gráficos simples.

- *Gödel, Escher y Bach*, de Douglas Hofstadter. Simplemente, si usted ha encontrado magia en la recursión, también la encontrará en éste best seller. Uno de los temas que trata Hofstadter es el de los "ciclos extraños", en los que los patrones evolucionan y ascienden hasta que se encuentran a sí mismos otra vez. La tesis de Hofstadter es que esos "ciclos extraños" son una parte esencial de lo que separa lo animado de lo inanimado. Él muestra patrones como éstos en la música de Bach, los cuadros de Escher y el teorema de incompletitud de Gödel.

Apéndice E

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft," which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

E.1. Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document," below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical, or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such,

"Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

E.2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in Section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

E.3. Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you

distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

E.4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.

- Preserve the section entitled "History," and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- In any section entitled "Acknowledgements" or "Dedications," preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.

- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements," provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you

may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

E.5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements,” and any sections entitled “Dedications.” You must delete all sections entitled “Endorsements.”

E.6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

E.7. Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate," and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.
If the Cover Text requirement of Section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

E.8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of Section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

E.9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense, or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

E.10. Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in

spirit to the present version, but may differ in detail to address new problems or concerns. See http:///www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License .ºr any later version.ªpplies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

E.11. Addendum: How to Use This License for Your Documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License."

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Apéndice F

Licencia de documentación libre de GNU

Versión 1.2, Noviembre 2002

Preámbulo

El propósito de esta licencia es permitir que un manual, libro de texto, u otro documento escrito sea libre en el sentido de libertad: asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, de manera

comercial o no. En segundo término, esta licencia proporciona al autor y al editor[2] una manera de obtener reconocimiento por su trabajo, sin que se le considere responsable de las modificaciones realizadas por otros.
Esta licencia es de tipo copyleft, lo que significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Complementa la Licencia Pública General de GNU, que es una licencia tipo copyleft diseñada para el software libre.
Hemos diseñado esta licencia para usarla en manuales de software libre, ya que el software libre necesita documentación libre: un programa libre debe venir con manuales que ofrezcan la mismas libertades que el software. Pero esta licencia no se limita a manuales de software; puede usarse para cualquier texto, sin tener en cuenta su temática o si se publica como libro impreso o no. Recomendamos esta licencia principalmente para trabajos cuyo fin sea instructivo o de referencia.

F.1. Aplicabilidad y definiciones

Esta licencia se aplica a cualquier manual u otro trabajo, en cualquier soporte, que contenga una nota del propietario de los derechos de autor que indique que puede ser distribuido bajo los términos de esta licencia. Tal nota garantiza en cualquier lugar del mundo, sin pago de derechos y sin límite de tiempo, el uso de dicho trabajo según las condiciones aquí estipuladas. En adelante la palabra Documento se referirá a cualquiera de dichos manuales o trabajos. Cualquier persona es un licenciatario y será referido como Usted. Usted acepta la licencia si copia. modifica o distribuye el trabajo de cualquier modo que requiera permiso según la ley de propiedad intelectual.
Una Versión Modificada del Documento significa cualquier trabajo que contenga el Documento o una porción del mismo, ya sea una copia literal o con modificaciones y/o traducciones a otro idioma.
Una Sección Secundaria es un apéndice con título o una sección preliminar del Documento que trata exclusivamente de la relación entre los autores o editores y el tema general del Documento (o temas relacionados) pero que no contiene nada que entre directamente en dicho tema general (por ejemplo, si el Documento es en parte un texto de matemáticas, una Sección Secundaria puede no explicar nada de matemáticas). La relación puede ser una conexión histórica con el tema o temas relacionados, o una opinión legal, comercial, filosófica, ética o política acerca de ellos.
Las Secciones Invariantes son ciertas Secciones Secundarias cuyos títulos son designados como Secciones Invariantes en la nota que indica que el documento es liberado bajo esta Licencia. Si una sección no entra en la definición de Secundaria, no puede designarse como Invariante. El documento puede no tener Secciones Invariantes. Si el Documento no identifica las Secciones Invariantes, es que no las

tiene.
Los Textos de Cubierta son ciertos pasajes cortos de texto que se listan como Textos de Cubierta Delantera o Textos de Cubierta Trasera en la nota que indica que el documento es liberado bajo esta Licencia. Un Texto de Cubierta Delantera puede tener como mucho 5 palabras, y uno de Cubierta Trasera puede tener hasta 25 palabras.
Una copia Transparente del Documento, significa una copia para lectura en máquina, representada en un formato cuya especificación está disponible al público en general, apto para que los contenidos puedan ser vistos y editados directamente con editores de texto genéricos o (para imágenes compuestas por puntos) con programas genéricos de manipulación de imágenes o (para dibujos) con algún editor de dibujos ampliamente disponible, y que sea adecuado como entrada para formateadores de texto o para su traducción automática a formatos adecuados para formateadores de texto. Una copia hecha en un formato definido como Transparente, pero cuyo marcaje o ausencia de él haya sido diseñado para impedir o dificultar modificaciones posteriores por parte de los lectores no es Transparente. Un formato de imagen no es Transparente si se usa para una cantidad de texto sustancial. Una copia que no es Transparente se denomina Opaca.
Como ejemplos de formatos adecuados para copias Transparentes están ASCII puro sin marcaje, formato de entrada de Texinfo, formato de entrada de LaTeX, SGML o XML usando una DTD disponible públicamente, y HTML, PostScript o PDF simples, que sigan los estándares y diseñados para que los modifiquen personas. Ejemplos de formatos de imagen transparentes son PNG, XCF y JPG. Los formatos Opacos incluyen formatos propietarios que pueden ser leídos y editados únicamente en procesadores de palabras propietarios, SGML o XML para los cuáles las DTD y/o herramientas de procesamiento no estén ampliamente disponibles, y HTML, PostScript o PDF generados por algunos procesadores de palabras sólo como salida.
La Portada significa, en un libro impreso, la página de título, más las páginas siguientes que sean necesarias para mantener legiblemente el material que esta Licencia requiere en la portada. Para trabajos en formatos que no tienen página de portada como tal, Portada significa el texto cercano a la aparición más prominente del título del trabajo, precediendo el comienzo del cuerpo del texto.
Una sección Titulada XYZ significa una parte del Documento cuyo título es precisamente XYZ o contiene XYZ entre paréntesis, a continuación de texto que traduce XYZ a otro idioma (aquí XYZ se refiere a nombres de sección específicos mencionados más abajo, como Agradecimientos, Dedicatorias , Aprobaciones o Historia. Conservar el Título de tal sección cuando se modifica el Documento significa que permanece una sección Titulada XYZ según esta definición .
El Documento puede incluir Limitaciones de Garantía cercanas a la nota donde se declara que al Documento se le aplica esta Licencia. Se considera que estas Limitaciones de Garantía están incluidas, por referencia, en la Licencia, pero sólo en

cuanto a limitaciones de garantía: cualquier otra implicación que estas Limitaciones de Garantía puedan tener es nula y no tiene efecto en el significado de esta Licencia.

F.2. Copia literal

Usted puede copiar y distribuir el Documento en cualquier soporte, sea en forma comercial o no, siempre y cuando esta Licencia, las notas de copyright y la nota que indica que esta Licencia se aplica al Documento se reproduzcan en todas las copias y que usted no añada ninguna otra condición a las expuestas en esta Licencia. Usted no puede usar medidas técnicas para obstruir o controlar la lectura o copia posterior de las copias que usted haga o distribuya. Sin embargo, usted puede aceptar compensación a cambio de las copias. Si distribuye un número suficientemente grande de copias también deberá seguir las condiciones de la sección 3.

Usted también puede prestar copias, bajo las mismas condiciones establecidas anteriormente, y puede exhibir copias públicamente.

F.3. Copiado en cantidad

Si publica copias impresas del Documento (o copias en soportes que tengan normalmente cubiertas impresas) que sobrepasen las 100, y la nota de licencia del Documento exige Textos de Cubierta, debe incluir las copias con cubiertas que lleven en forma clara y legible todos esos Textos de Cubierta: Textos de Cubierta Delantera en la cubierta delantera y Textos de Cubierta Trasera en la cubierta trasera. Ambas cubiertas deben identificarlo a Usted clara y legiblemente como editor de tales copias. La cubierta debe mostrar el título completo con todas las palabras igualmente prominentes y visibles. Además puede añadir otro material en las cubiertas. Las copias con cambios limitados a las cubiertas, siempre que conserven el título del Documento y satisfagan estas condiciones, pueden considerarse como copias literales.

Si los textos requeridos para la cubierta son muy voluminosos para que ajusten legiblemente, debe colocar los primeros (tantos como sea razonable colocar) en la verdadera cubierta y situar el resto en páginas adyacentes.

Si Usted publica o distribuye copias Opacas del Documento cuya cantidad exceda las 100, debe incluir una copia Transparente, que pueda ser leída por una máquina, con cada copia Opaca, o bien mostrar, en cada copia Opaca, una dirección de red donde cualquier usuario de la misma tenga acceso por medio de protocolos públicos y estandarizados a una copia Transparente del Documento completa, sin material adicional. Si usted hace uso de la última opción, deberá tomar las medidas

necesarias, cuando comience la distribución de las copias Opacas en cantidad, para asegurar que esta copia Transparente permanecerá accesible en el sitio establecido por lo menos un año después de la última vez que distribuya una copia Opaca de esa edición al público (directamente o a través de sus agentes o distribuidores).
Se solicita, aunque no es requisito, que se ponga en contacto con los autores del Documento antes de redistribuir gran número de copias, para darles la oportunidad de que le proporcionen una versión actualizada del Documento.

F.4. Modificaciones

Puede copiar y distribuir una Versión Modificada del Documento bajo las condiciones de las secciones 2 y 3 anteriores, siempre que usted libere la Versión Modificada bajo esta misma Licencia, con la Versión Modificada haciendo el rol del Documento, por lo tanto dando licencia de distribución y modificación de la Versión Modificada a quienquiera posea una copia de la misma. Además, debe hacer lo siguiente en la Versión Modificada:

- Usar en la Portada (y en las cubiertas, si hay alguna) un título distinto al del Documento y de sus versiones anteriores (que deberían, si hay alguna, estar listadas en la sección de Historia del Documento). Puede usar el mismo título de versiones anteriores al original siempre y cuando quien las publicó originalmente otorgue permiso.
- Listar en la Portada, como autores, una o más personas o entidades responsables de la autoría de las modificaciones de la Versión Modificada, junto con por lo menos cinco de los autores principales del Documento (todos sus autores principales, si hay menos de cinco), a menos que le eximan de tal requisito.
- Mostrar en la Portada como editor el nombre del editor de la Versión Modificada.
- Conservar todas las notas de copyright del Documento.
- Añadir una nota de copyright apropiada a sus modificaciones, adyacente a las otras notas de copyright.
- Incluir, inmediatamente después de las notas de copyright, una nota de licencia dando el permiso para usar la Versión Modificada bajo los términos de esta Licencia, como se muestra en la Adenda al final de este documento.
- Conservar en esa nota de licencia el listado completo de las Secciones Invariantes y de los Textos de Cubierta que sean requeridos en la nota de Licencia del Documento original.

- Incluir una copia sin modificación de esta Licencia.
- Conservar la sección Titulada Historia, conservar su Título y añadirle un elemento que declare al menos el título, el año, los nuevos autores y el editor de la Versión Modificada, tal como figuran en la Portada. Si no hay una sección Titulada Historia en el Documento, crear una estableciendo el título, el año, los autores y el editor del Documento, tal como figuran en su Portada, añadiendo además un elemento describiendo la Versión Modificada, como se estableció en la oración anterior.
- Conservar la dirección en red, si la hay, dada en el Documento para el acceso público a una copia Transparente del mismo, así como las otras direcciones de red dadas en el Documento para versiones anteriores en las que estuviese basado. Pueden ubicarse en la sección Historia. Se puede omitir la ubicación en red de un trabajo que haya sido publicado por lo menos cuatro años antes que el Documento mismo, o si el editor original de dicha versión da permiso.
- En cualquier sección Titulada Agradecimientos o Dedicatorias, Conservar el Título de la sección y conservar en ella toda la sustancia y el tono de los agradecimientos y/o dedicatorias incluidas por cada contribuyente.
- Conservar todas las Secciones Invariantes del Documento, sin alterar su texto ni sus títulos. Números de sección o el equivalente no son considerados parte de los títulos de la sección.
- Borrar cualquier sección titulada Aprobaciones. Tales secciones no pueden estar incluidas en las Versiones Modificadas.
- No cambiar el título de ninguna sección existente a Aprobaciones ni a uno que entre en conflicto con el de alguna Sección Invariante.
- Conservar todas las Limitaciones de Garantía.

Si la Versión Modificada incluye secciones o apéndices nuevos que califiquen como Secciones Secundarias y contienen material no copiado del Documento, puede opcionalmente designar algunas o todas esas secciones como invariantes. Para hacerlo, añada sus títulos a la lista de Secciones Invariantes en la nota de licencia de la Versión Modificada. Tales títulos deben ser distintos de cualquier otro título de sección.
Puede añadir una sección titulada Aprobaciones, siempre que contenga únicamente aprobaciones de su Versión Modificada por otras fuentes –por ejemplo, observaciones de peritos o que el texto ha sido aprobado por una organización como la definición oficial de un estándar.
Puede añadir un pasaje de hasta cinco palabras como Texto de Cubierta Delantera y un pasaje de hasta 25 palabras como Texto de Cubierta Trasera en la Versión

Modificada. Una entidad solo puede añadir (o hacer que se añada) un pasaje al Texto de Cubierta Delantera y uno al de Cubierta Trasera. Si el Documento ya incluye un textos de cubiertas añadidos previamente por usted o por la misma entidad que usted representa, usted no puede añadir otro; pero puede reemplazar el anterior, con permiso explícito del editor que agregó el texto anterior.
Con esta Licencia ni los autores ni los editores del Documento dan permiso para usar sus nombres para publicidad ni para asegurar o implicar aprobación de cualquier Versión Modificada.

F.5. Combinación de documentos

Usted puede combinar el Documento con otros documentos liberados bajo esta Licencia, bajo los términos definidos en la sección 4 anterior para versiones modificadas, siempre que incluya en la combinación todas las Secciones Invariantes de todos los documentos originales, sin modificar, listadas todas como Secciones Invariantes del trabajo combinado en su nota de licencia. Así mismo debe incluir la Limitación de Garantía.
El trabajo combinado necesita contener solamente una copia de esta Licencia, y puede reemplazar varias Secciones Invariantes idénticas por una sola copia. Si hay varias Secciones Invariantes con el mismo nombre pero con contenidos diferentes, haga el título de cada una de estas secciones único añadiéndole al final del mismo, entre paréntesis, el nombre del autor o editor original de esa sección, si es conocido, o si no, un número único. Haga el mismo ajuste a los títulos de sección en la lista de Secciones Invariantes de la nota de licencia del trabajo combinado.
En la combinación, debe combinar cualquier sección Titulada Historia de los documentos originales, formando una sección Titulada Historia; de la misma forma combine cualquier sección Titulada Agradecimientos, y cualquier sección Titulada Dedicatorias. Debe borrar todas las secciones tituladas Aprobaciones.

F.6. Colecciones de documentos

Puede hacer una colección que conste del Documento y de otros documentos liberados bajo esta Licencia, y reemplazar las copias individuales de esta Licencia en todos los documentos por una sola copia que esté incluida en la colección, siempre que siga las reglas de esta Licencia para cada copia literal de cada uno de los documentos en cualquiera de los demás aspectos.
Puede extraer un solo documento de una de tales colecciones y distribuirlo individualmente bajo esta Licencia, siempre que inserte una copia de esta Licencia en el documento extraído, y siga esta Licencia en todos los demás aspectos relativos a la copia literal de dicho documento.

F.7. Agregación con trabajos independientes

Una recopilación que conste del Documento o sus derivados y de otros documentos o trabajos separados e independientes, en cualquier soporte de almacenamiento o distribución, se denomina un agregado si el copyright resultante de la compilación no se usa para limitar los derechos de los usuarios de la misma más allá de lo que los de los trabajos individuales permiten. Cuando el Documento se incluye en un agregado, esta Licencia no se aplica a otros trabajos del agregado que no sean en sí mismos derivados del Documento.
Si el requisito de la sección 3 sobre el Texto de Cubierta es aplicable a estas copias del Documento y el Documento es menor que la mitad del agregado entero, los Textos de Cubierta del Documento pueden colocarse en cubiertas que enmarquen solamente el Documento dentro del agregado, o el equivalente electrónico de las cubiertas si el documento está en forma electrónica. En caso contrario deben aparecer en cubiertas impresas enmarcando todo el agregado.

F.8. Traducción

La Traducción es considerada como un tipo de modificación, por lo que usted puede distribuir traducciones del Documento bajo los términos de la sección 4. El reemplazo de las Secciones Invariantes con traducciones requiere permiso especial de los dueños de derecho de autor, pero usted puede añadir traducciones de algunas o todas las Secciones Invariantes a las versiones originales de las mismas. Puede incluir una traducción de esta Licencia, de todas las notas de licencia del documento, así como de las Limitaciones de Garantía, siempre que incluya también la versión en Inglés de esta Licencia y las versiones originales de las notas de licencia y Limitaciones de Garantía. En caso de desacuerdo entre la traducción y la versión original en Inglés de esta Licencia, la nota de licencia o la limitación de garantía, la versión original en Inglés prevalecerá.
Si una sección del Documento está Titulada Agradecimientos, Dedicatorias o Historia el requisito (sección 4) de Conservar su Título (Sección 1) requerirá, típicamente, cambiar su título.

F.9. Terminación

Usted no puede copiar, modificar, sublicenciar o distribuir el Documento salvo por lo permitido expresamente por esta Licencia. Cualquier otro intento de copia, modificación, sublicenciamiento o distribución del Documento es nulo, y dará por terminados automáticamente sus derechos bajo esa Licencia. Sin embargo, los terceros que hayan recibido copias, o derechos, de usted bajo esta Licencia no verán terminadas sus licencias, siempre que permanezcan en total conformidad con ella.

F.10. Revisiones futuras de esta licencia

De vez en cuando la Free Software Foundation puede publicar versiones nuevas y revisadas de la Licencia de Documentación Libre GNU. Tales versiones nuevas serán similares en espíritu a la presente versión, pero pueden diferir en detalles para solucionar nuevos problemas o intereses. Vea http://www.gnu.org/copyleft/.
Cada versión de la Licencia tiene un número de versión que la distingue. Si el Documento especifica que se aplica una versión numerada en particular de esta licencia o cualquier versión posterior, usted tiene la opción de seguir los términos y condiciones de la versión especificada o cualquiera posterior que haya sido publicada (no como borrador) por la Free Software Foundation. Si el Documento no especifica un número de versión de esta Licencia, puede escoger cualquier versión que haya sido publicada (no como borrador) por la Free Software Foundation.

F.11. ADENDA: Cómo usar esta Licencia en sus documentos

Para usar esta licencia en un documento que usted haya escrito, incluya una copia de la Licencia en el documento y ponga el siguiente copyright y nota de licencia justo después de la página de título:

> Copyright (c) AÑO SU NOMBRE. Se concede permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre de GNU, Versión 1.2 o cualquier otra versión posterior publicada por la Free Software Foundation; sin Secciones Invariantes ni Textos de Cubierta Delantera ni Textos de Cubierta Trasera. Una copia de la licencia está incluida en la sección titulada GNU Free Documentation License.

Si tiene Secciones Invariantes, Textos de Cubierta Delantera y Textos de Cubierta Trasera, reemplace la frase sin ... Trasera por esto:

> siendo las Secciones Invariantes LISTE SUS TÍTULOS, siendo los Textos de Cubierta Delantera LISTAR, y siendo sus Textos de Cubierta Trasera LISTAR.

Si tiene Secciones Invariantes sin Textos de Cubierta o cualquier otra combinación de los tres, mezcle ambas alternativas para adaptarse a la situación.
Si su documento contiene ejemplos de código de programa no triviales, recomendamos liberar estos ejemplos en paralelo bajo la licencia de software libre que usted elija, como la Licencia Pública General de GNU (GNU General Public License), para permitir su uso en software libre.

Notas
[1] Ésta es la traducción del Copyright de la Licencia, no es el Copyright de esta traducción no autorizada.
[2] La licencia original dice publisher, que es, estrictamente, quien publica, diferente de editor, que es más bien quien prepara un texto para publicar. En castellano editor se usa para ambas cosas.
[3] En sentido estricto esta licencia parece exigir que los títulos sean exactamente Acknowledgements, Dedications, Endorsements e History, en inglés.

Índice analítico